(2)

NORTH ATLANTIC TREATY ORGANIZATION
DEFENCE RESEARCH GROUP

TECHNICAL PROCEEDINGS
AC/243(Panel 11)TP/1

SYMPOSIUM ON MILITARY INFORMATION SYSTEMS ENGINEERING

PROCEEDINGS

Panel 11 on Information Processing Technology

CONSEIL DE L'ATLANTIQUE NORD
NORTH ATLANTIC COUNCIL

UNCLASSIFIED/UNLIMITED

ORIGINAL: ENGLISH
27 December 1991

TECHNICAL PROCEEDINGS
AC/243(Panel 11)TP/1

DEFENCE RESEARCH GROUP

PANEL 11 ON INFORMATION PROCESSING TECHNOLOGY

Technical Proceedings of the Symposium on Military Information Systems Engineering

Note by the Secretary

These are the Technical Proceedings of a Symposium on Military Information Systems Engineering. It was organized by Panel 11 on Information Processing Technology. The Executive Summary of these Proceedings ("Yellow Pages") will be distributed under reference AC/243-N/347 dated 13 January 1992.

(Signed) Dr. J. VERMOREL
Defence Research Section

NATO,
1110 Brussels.

NATO UNCLASSIFIED

REPORT DOCUMENTATION PAGE

1. Recipient's Reference:	2. Further Reference:
3. Originator's Reference: AC/243(Panel 11)TP/1	4. Security Classification: UNCLASSIFIED/UNLIMITED 5. Date: 27 DEC 91 — 6. Total Pages: 235 p.

7. Title (NU):
SYMPOSIUM ON MILITARY INFORMATION SYSTEMS ENGINEERING

8. Presented at:
RSRE, Malvern, UK 8-10 May 1990

9. Author's/Editor's:
S. Bond et al.

10. Author(s)/Editor(s) Address:
RSRE Malvern
St. Andrews Road
Malvern
Worcestershire WR14 3PS
United Kingdom

11. NATO Staff Point of Contact:
Defence Research Section
NATO Headquarters
B-1110 Brussels
Belgium
(Not a Distribution Centre)

13. Keywords/Descriptors:
KNOWLEDGE-BASED SYSTEMS, SOFTWARE TECHNOLOGY, C3I SYSTEMS DEVELOPMENT, LIFE-CYCLE MODELS, FORMAL METHODS

14. Abstract:

Panel 11 organized a symposium for information exchange and discussion of current research in information processing technology and its application to the engineering of military information systems. Four sessions were held:
(a) Engineering for the System Life-Cycle,
(b) Knowledge-Based Systems,
(c) Software Technology,
(d) C3I Systems Development.
As a main conclusion, system procurement practices allowing for incremental and evolutionary development would bring significant benefit.

OUTLINE OF CHAPTERS

CHAPTER 0

EXECUTIVE SUMMARY

0.1 Summary of the Symposium

i. The symposium provided a forum for information exchange and discussion of current research in information processing technology and its application to the engineering of military information systems.

ii. The objectives were:

(a) To assess the problems of engineering information processing systems for military use

(b) To identify emerging technologies and techniques for system engineering and evaluate the benefits offered

(c) To determine the research needed to facilitate the introduction and application of beneficial techniques to the engineering of future military systems.

iii. The following summary outlines the topics considered in the presentations and the major points of the discussions. Four sessions were held on the topics of:

(a) Engineering for the System Life-Cycle

(b) Knowledge-Based Systems

(c) Software Technology

(d) C3I Systems Development

0.2 Engineering for the System Life-Cycle

iv. Six papers were presented covering prototyping, software evolution, object-oriented and transformational techniques for system specification, formal methods and the spiral model for life-cycle system development. Some of the questions addressed were (1) the purpose, potential benefits and problems of prototyping and (2) the identification of various models for system development, and their suitability and application.

v. The following observations were made. There is more than one model for life-cycle systems development. They range from the traditional waterfall model, which is understood from a conceptual and contracting point of view but which is not effective for systems addressing vague or evolving requirements, through models which involve prototyping to the spiral model which assumes a number of iterations and can develop at each iteration a system that contains subsystems at differing levels of abstraction. Although spiral-type life-cycle models appear most attractive for military systems, there has been little practical experience with them and there needs to be a cultural change to allow them to be addressed by the procurement agencies and to be managed effectively.

vi. The purpose of prototyping is to clarify requirements, to solidify specifications, and to give an early "look and feel" for the system users. This leads to significant user involvement in the early stages and better user understanding and acceptance of the developed system. However it can lead to design decisions being made too early in the development cycle. There can be a perception that the prototype is the real system and that further (costly) engineering need not take place. This is a tendency that should be resisted as prototyping tends to concentrate on the mainstream requirements rather than the exceptions.

vii. The session identified formal methods and object-oriented paradigms as emerging technologies addressing the requirements and specification phases. Formal methods give the potential for error-free transformations from system specification to implementation. The object-oriented methods support rapid prototyping.

0.3 Knowledge-based Systems

viii. Two papers focused on the use of Artificial Intelligence techniques for the automation of software development, and the quality control of expert systems for operational use.

ix. The goal of automating software development from an initial (formal) specification into executable code is ambitious and long-term. For data handling, the research addresses data type design rather than algorithms. Real-time issues are not yet considered.

x. Validation of expert systems intended for operational use is essential. Traditional development and quality control models do not map well on to expert systems. Reliability and maintainability of the knowledge base are recognized as the principal quality criteria.

xi. Future operational use of knowledge-based systems should not be restricted to stand-alone sub-systems. They could offer a higher level of help in decision making within command and control systems. This raises questions such as (1) how to connect a knowledge base to the rest of the system (2) how to identify and bound the necessary knowledge and (3) how to collect the knowledge that exists.

0.4 Software Technology

xii. Three papers identified promising technologies for the improved development of software systems: a universal intermediate language (TDF) as an architecture-independent and programming language-independent format for program; the development language DEVA, aimed at the formal development of software objects which would be "correct by construction"; and some aspects of the ARCADIA research programme.

xiii. The potential impact of these technologies was described. A universal intermediate language is an approach to software distribution and portability. The introduction of an ANDF (Architecture Neutral Distribution Format) is expected to increase software re-use, encouraging the sale of software components which would all be distributed in ANDF for installation on a variety of computer architectures.

xiv. The use of formal methods was seen as conferring advantages in obtaining correct programs and to be fruitful in the context of software re-use as a means of solving sets of similar problems. Potential problems are its scalability for large systems and possible difficulties of understanding.

xv. The ARCADIA approach attempts to break down the distinction between programming language facilities and database facilities by providing strong typing integrated with persistent data. Discussion included comparison with the PCTE and CAIS interface specifications. Both PCTE and CAIS use a separate database management system. These systems were observed to be based on 1970's research, whereas persistent systems were aimed at the next generation.

0.5 C3I Systems Development

xvi. Four papers ranged from the conceptual to the experimental. An experimental implementation was reported of a general architecture for a distributed command and control information system using the ISO/OSI Reference Model, the Ada language and commercial off-the-shelf software (COTS). This highlighted the effort still needed to adapt and integrate COTS with security components and specific applications such as expert system components.

xvii. A paper on the development approach for C3I illustrated the benefits offered by object oriented languages in meeting characteristics of C3I systems such as interoperability, security, integrity and testability, and the need for the system to evolve to be able to handle changes.

xviii. A case study of the development of Army C3I systems described activities over 20 years. Extensive system modelling was carried out to define the hardware and software architectures with respect to redundancy, performance, survivability and functionality. The subsequent paper also developed the theme of survivability. Survivability may be achieved in the most cost-effective way through replication of function and dispersion. Experiments looked at multi-media communications capability, the evaluation of re-connection strategies and the use of advanced technologies for creating a strategy planning system in a highly stressed environment.

xix. The discussion again emphasised that the role of the user during system development is essential. The development method must be chosen accordingly and prototyping seems a promising approach. Military users are conservative in terms of technology, so high level quality prototypes are essential for acceptance.

0.6 Main Conclusions

xx. Several emerging technologies were identified as having potential benefit for future military systems. Various technical problem areas were agreed to need continuing research. In addition, the discussions identified significant problems concerned with the mapping of new system development methods and life-cycle models to procurement practices.

xxi. Emerging technologies include formal methods, prototyping, object-oriented programming, knowledge-based systems, persistent data systems and techniques for software portability. Many different and promising paradigms are being used. However, the integration of the different development paradigms can cause problems, particularly in the approach to system integration. Formal methods are currently immature but will become an important tool in the longer term. This was identified as one of the topics for future research activities.

xxii. Further work is needed to define the role of prototyping and its relation to the system development model. An important element for study is the means of transition from prototype to engineered operational system. The infrastructure for operational systems seems less well developed than that

currently being applied within the prototyping environments. To ease the transition further work should identify improvements for the operational infrastructure and how they might be achieved.

xxiii. The development of C3I systems requires specific approaches to solve problems such as the integration process, the use of commercially available hardware and/or software, the re-usability of software and more specifically military issues such as security, interoperability and survivability. This type of system is very complex and requires a large variety of engineering disciplines. Research and experimentation has shown the value of an incremental and evolutionary development process including prototyping. The issues raised by this are not only technical but concern the adaptation of procurement policies to the emerging development technology. Such issues were not considered as direct research problems.

0.7 Major Recommendations

xxiv. The symposium identified several areas to be considered for future activities of Panel 11, including Formal Methods, Software re-usability, portability, and interoperability, the use of Commercial off-the-shelf software and Knowledge Engineering in C3I.

xxv. The symposium discussions found that it was difficult to evaluate the benefits offered by emerging technology in the context of current procurement practice. The involvement of the military sponsor and the future operational user from an early stage is essential in incremental and iterative development. No recognized mechanisms exist to ensure this involvement.

xxvi. Thus, in considering the impact of emerging technology the problems are not entirely technical. Many are connected with the acquisition culture in defence which is strongly connected to a waterfall model of systems development. Current procurement practice was felt to legislate against the benefits expected from incremental and evolutionary development.

xxvii. There is an urgent need for a transition strategy, including consideration of changed procurement practices, in order to allow full use of beneficial emerging technology in the engineering of future military information systems.

0.8 Military Implications

xxviii. Military operations place increasing reliance on the use of software based systems for military operations. Systems in which software has a critical part range from embedded weapons systems to large distributed C3I.

The characteristics required of military C3I systems include security, reliability, interoperability and survivability. This implies certain properties of the system architecture such as distribution and reconfigurability. System development is carried out against a background of rapidly developing commercial hardware technology offering considerable price and performance benefits. However, commercial developments in systems engineering have not kept pace with this hardware technology improvement and do not adequately address specific military requirements.

xxix. System procurement practices allowing for incremental and evolutionary development would bring significant benefit. These methods help to clarify requirements. They allow the incorporation of commercial off-the-shelf software in appropriate subsystems. They allow utilization of relevant standards including emerging Open Systems. Increasing re-use of software and the ability to take timely advantage of commercial investment will be essential for the cost-effective development and interoperability of future military systems.

CHAPTER 1

1 OBJECTIVES OF THE SYMPOSIUM

1.1 Introduction

1. The symposium on Military Information Systems Engineering held by Panel 11 at RSRE, Malvern, UK on 8-10th May 1990 aimed to provide a forum for information exchange and discus[illegible]current research relating to digital computing and information processing technology for defence systems, and to identify areas where future co-operative activities might be undertaken. The initial call for papers intentionally had a wide scope. It suggested technology topics such as high performance systems, massively parallel architectures, expert system techniques, high integrity systems, databases for system design, project support environments and software prototyping techniques together with considerations of the system life-cycle from design to maintenance and the specific requirements of military information systems. Most of the abstracts received dealt with software engineering, system development models and the characteristics of large command, control and communications information systems. The symposium was accordingly structured around these major topics.

1.2 Organization of the Symposium

2. The invited presentations were arranged in four sessions:

Engineering for the System Life-Cycle
Software Technology
Knowledge-Based Systems
C3I Systems Development

3. Considerable time for discussion was allowed after each group of papers and the symposium concluded with a final session addressing the questions and themes raised by the presentations. This resulted in a "workshop-style" event, which was attended by 42 participants from ten nations and NATO agencies.

1.3 Opening Session

4. The participants were welcomed to RSRE, Malvern, by Dr Boyd Burgess, Head of Communications and Computing Group, RSRE on behalf of Director, RSRE. Dr Burgess described RSRE's position as one of the four UK military Defence Research Establishments, concerned with basic radar research, command and control systems, communications and generic electronics technology

with application to defence systems. Communications and Computing Group is the UK MOD centre for computing systems engineering and secure distributed information systems research.

5. The Chairman of Panel 11, ICA J P Crestin responded, thanking RSRE for the preparation of the Symposium and the participants for their interest. He described the role of the Panel, which is the most recent panel of the DRG. It was created 4-5 years ago, and has 3 sub groups at present: RSG1 on Distributed Systems Design Methodology, RSG2 on Trustworthy Systems and an Exploratory Group on Software Engineering. Panel 11 works in a demanding environment. The importance of software to defence systems is increasing. However, commercial efforts surpass the military and it is impossible to cover the whole field owing to the breadth of civil developments. A definition of the specificity of the military requirement is needed to determine which research activities should be undertaken. The Symposium would contribute to the definition of future topics for the work of Panel 11.

6. The Symposium Director, Miss S G Bond, then presented the detailed objectives for the symposium:

- To assess the problems of engineering information processing systems for military use

- To identify emerging technologies and techniques for systems engineering and evaluate the benefits offered

- To determine the research needed to facilitate the introduction and application of beneficial techniques to the engineering of future military systems.

7. Certain considerations should be addressed in discussion to meet these objectives. The workshop needed to

- determine the specific characteristics of military information processing systems

- review models for the development of information systems

- assess the impact of emerging techniques on system design and development

- assess the problems of the application of these techniques to the engineering of future military systems.

CHAPTER 2

2.0 ENGINEERING FOR THE SYSTEM LIFE CYCLE

Session Chairman: V K Taylor (Canada)
Rapporteur: T A D White (UK)

2.1 Introduction

8. The "Engineering for the System Life Cycle" sessions contained six papers and extended discussions with topics ranging from prototyping, software evolution, the Spiral model to formal methods.

9. Some of the questions addressed were:

- What are the models for life-cycle system development? Is one model suitable for the development of all software systems?
- What is the purpose of prototyping? What benefits are to be gained, and what problems are encountered?
- What technologies are emerging for the development of systems, and what benefits do they offer?

2.2 Summary of Presentations

10. M Looney (UK) in his paper (with I Sinclair) on "Management and Control of Prototyping as part of the Development Life Cycle" identified the importance of iterative development of requirements, testing against real users needs by prototyping. What is prototyped is the system structure and basic functionality. This approach gives savings through life cycle coordination, reduces risk through early visibility of the system for comparison against perceived requirements and results in greater acceptability of the developed system. The potential disadvantage lies in the short term cost of prototyping and the potential paradox that the user may want the "skeletal" prototype to be his system even though it is still merely the foundation for the operational system.

11. B Barry (CA) presented a "Case Study in Object Oriented System Engineering" in which AMEP, a prototype for an ESM signal processor was discussed. It was estimated that the incremental, evolutionary prototyping approach to "growing" more functions and capabilities has led to a user accepted system that will be transitioned to industry for development. The object oriented prototyping approach was taken because of the vague and

incomplete original requirements and the necessity to "build cheap" through changing requirements. A significant lesson learned was that the object oriented approach works and it is important to use a programming system that works with ones method, not against it.

12. M. Gentleman (CA), in his presentation "Engineering the Process of Software Evolution" highlighted the problems and potential for software reuse. He introduced the concept of a "family of programs" in which the top down design approach is not sensible. Shared components must be considered first which implies a bottom-up design strategy. He considered that software reuse extends to architectures, algorithms, data structures, file structures, software components, databases, documentation, test procedures and user interfaces. DoD Mil Std 2167A was considered very unhelpful for encouraging reuse. The standard contractual process makes the problem worse - lowest bidder and "cost plus" contracting do not encourage reuse. Further, the specification of change evolution and tool support should be part of the contracting process.

13. D Fikkert (NL) in his paper "The Spiral Model, Some Problems, Many Solutions?" introduced the model and noted that experimental use of the Spiral model is needed along with exchange of experience. Also, well defined interfaces between project management, quality management and assurance, life cycle, technical development and procurement are needed. Change is inevitable and requires planning.

14. C Lewis (UK) in his paper (with B Ratcliff) "Transformational Implementation of JSD Specifications in Smalltalk-80" discussed semantics preserving transformations from the JSD framework into the object oriented data abstraction and encapsulation paradigm that supports software reuse. The rationale for this approach is to prototype specifications.

15. P Place (US) with W Wood contributed a paper "Formal Specification and Requirements" in which recent work at the Software Engineering Institute was discussed. The presentation concluded that formal methods are particularly useful in the system specification phases and helped to identify ambiguities and alternatives. Effective communication among team members was enhanced by the approach. However, the presentation noted that formal methods will not make the requirements correct; it will only help in ensuring that the requirements stated are reflected in the underlying specification with a commensurate potential for consistent design and implementation from the requirements.

12.3 Discussion

16. There was considerable discussion following the papers and at the end of the session. The following gives the flavour of the comments made.

17. Prototyping captures requirements. The prototype exercises requirements and allows the result to be validated by the customer. Usually the prototype models the main functions and perhaps the logical structure of the overall system. The man-machine interface is also a candidate for prototyping. There is a need for heavy user involvement but you must be prepared for the prototyping paradox - the system looks good and is wanted by the customer but in fact it is only a shell masquerading as the total system. Not all requirements will have been met. However, it was pointed out that if the prototype "works", then you do have the basic requirement, and if the prototype "fails" then lessons have been learned with regard to the understanding of the requirements. From this point of view, whether successful or a failure, prototyping is likely to be money well spent. It is still not clear how the "essence" of a prototype is captured to allow it to be a description of the required system suitable to serve in a contract. It was observed that seeing one prototype only allows you to envisage improvements to one approach, not to consider radically different approaches.

18. With respect to software reuse, it was observed that in the present climate, reuse benefits the user but not the developer. How do you give incentive to contractors to build reusable code? It is clear to commercial developers that software reuse is essential and within a corporation there is no contractual difficulty. However, defence contracting appears different and there need to be some ways of building software reuse into contracts. If you can get around the contracting issue, then you can address the specification of components including the performance requirements and testing procedures. This is needed for reusability. However, current contracting culture makes reuse expensive. With reusable software performance measurement becomes part of the process. The technical standards currently in vogue do not advance the cause of software reuse. Language does not necessarily solve problems; Ada is no better or worse for sharing and reuse than other commonly used languages. Integrated Project Support Environments, and present standardization efforts do not appear to help re-use. Program portability theory is not improving and people do not understand the issues involved.

19. There was an interesting interchange about the use of formal methods. Safety/Liveness properties were assessed using proof theoretic techniques in a manual way, although there was an automated toolset for temporal logic. The assessment was not very detailed as there was not much

time available for this aspect. Time variant CSP was looked at but not used as the intent was to use "mainstream" techniques. The specifications were not just mathematically based but also included text. The specification reviewers, who were not formal methods experts, appeared able to relate to specifications in this form. Formal methods help to clarify requirements. The system developer should be given both the formalism plus text. Although the actual process of going from specification to code is not clear, there are definite advantages to clarifying requirements. At the present state of formal methods development, the specification of interrupts is complicated but tractable.

2.4 Conclusion

20. The session on "Engineering for the System Life Cycle" was wide ranging and the following observations can be made. There is more than one model for Life Cycle systems development. They range from the traditional waterfall model, which is understood from a conceptual and contracting point of view but which is not effective for systems addressing vague or evolving requirements, through models which involve prototyping to clarify requirements and solidify specifications, to the Spiral model which assumes a number of iterations and can develop at each iteration systems that contain subsystems at differing levels of abstraction.

21. All participants essentially agreed that the Waterfall model of system development, although easy to contract, does not work for evolutionary or original systems but may only have application to systems that are essentially replications of earlier systems with a well defined and static external operating environment. Speakers advocated the advantages of prototyping to evolve either system requirements or else the complete system to a state where it is safe to engineer with a reasonable expectation that the system will be accepted by the user. Although the Spiral type life cycle models appear most attractive for military systems, there has been little practical experience with them and there needs to be a cultural change to allow them to be addressed by the procurement agencies and to be managed effectively.

22. A purpose for prototyping is to give an early "look and feel" for system users. This needs significant user involvement in the early stages and leads to better user understanding and acceptance of the developed system. Prototyping will help identify requirements deficiencies. However it can lead to design decisions being made too early in the development cycle. There can be a perception that the prototype is the real system and that further (costly) engineering need not take place. This is a tendency that should be resisted as prototyping tends to concentrate on the mainstream requirements

and concepts rather than the exceptions. One interesting idea is that the prototype can be considered part of the documentation needed for the engineering of the implemented systems.

23 The session identified formal methods and object-oriented paradigms as technologies emerging for the development of systems, addressing the requirements and specification phases. Formal methods help clarify requirements and give the potential for error-free transformations from system specification to implementation. The object-oriented methods support rapid prototyping.

CHAPTER 3

3. KNOWLEDGE BASED SYSTEMS

Session Chairman: ICA B Vors (France)
Rapporteur: Dr B M Barry (Canada)

3.1 Summary of Presentations

24. This session did not attempt to address the whole scope of Artificial Intelligence in military systems but focused on two aspects related to system engineering: automating software development and the quality of expert systems.

25. In his paper "Automating the development of Software" Douglas R Smith (US) presented KIDS (Kestrel Interactive Development System), an experimental environment which offers a "derivation process" for software development. Formal specifications - considered as a pre-requisite - are interactively developed into code through a series of correctness-preserving transformations. Tools for performing algorithm design, deductive inference, program simplification, finite differencing optimizations, data structure refinement and others are available to the program developer.

26. The presentation also included a brief discussion of the application of these ideas to maintenance, prototyping and re-use (with stress on a broad view of re-use, not just code, but program knowledge and design decisions).

27. The "Quality of Expert Systems" by Michael Perre (NL) described the integration of database theory and artificial intelligence as a step in the direction of a better quality control of expert systems. The approach was motivated by the perception that 'intelligent' modules are not developed with the same attention to rigorous quality control as other subsystems. A distinction was made between verification (has a specification been correctly implemented) and validation (is the specification itself correct).

28. It was proposed that a number of techniques can be borrowed from other areas to address the quality issue. These include structured development methods, test and evaluation strategies, modular design and some aspects of database theory. Deficiencies in these techniques when applied to expert systems were also noted. These points were illustrated by a description of DAMOCLES, a damage monitoring and control system.

3.2 Discussion

29. The discussion recognized that the goal of automating software development by the transformation of the initial formal specification into executable code is ambitious and long-term. For example, the research addresses data type design rather than data handling algorithms, and real-time issues are not yet considered.

30. Validation of Expert Systems intended for operational use was considered essential. However, traditional development and quality control models do not map well onto expert systems. Reliability and maintainability of the knowledge base are recognized as the principal quality criteria. The evaluation and testing of expert systems is an under-developed field of study. Often it is not possible to test a system against an objective standard and methods focused on a structured generation of expert system test cases are not yet available.

31. Knowledge engineering could offer useful enhancements to system development, for example in the design process as a link between the technical and user points of view to elucidate requirements, or as a help when choosing between design options. The subject is broad.

32. Future operational use of knowledge based systems should not be restricted to stand-alone subsystems. They could offer a higher level of help in decision making within command and control systems. This raises questions such as:

- how to connect a knowledge base to the rest of the system
- how to identify and bond the necessary knowledge
- how to collect knowledge when it exists.

33. These are real problems, not within the scope of this symposium, but a subject that should be addressed in the future.

CHAPTER 4

4. SOFTWARE TECHNOLOGY

Session Chairman: Charles J Holland (USA)
Rapporteur: Mrs M Stanley (UK)

4.1 Summary of presentations and discussion

34. The Software Technology session contained three papers identifying promising technologies for the improved development of software systems.

35. Dr Peeling (UK) in his paper "Ten15 - A High Integrity Kernel for Software Engineering Applications" reviewed the RSRE development of TDF (The Ten15 Distribution Format) as a universal intermediate language to satisfy the needs for software distribution and portability. It is a language independent, architecture independent format comparable to a compiler intermediate language. The introduction of an ANDF (Architecture Neutral Distribution Format) is expected to increase software reuse, encouraging the sale of software components, all of which use the ANDF.

36. In the discussion period, two reasons were put forward for the rejection in the past of intermediate languages:

(i) debugging problems and the difficulty of recovering source

(ii) the problem of mixed language integration, such as in compatible calling conventions.

37. The debugging problem has been dealt with by retaining the name/value bindings in the compiler code. The calling convention problem can be dealt with by adopting the calling convention of the target architecture assembler. Compilers for a target machine normally adopt the convention for integration with the target operating system, so this is not a penalty.

38. There is an on-going debate as to who should provide the installer for an ANDF (translating ANDF program for a target architecture). Should the installer be provided as part of the ANDF, written using automatic retargeting techniques but giving relatively low performance, or by the hardware vendors? The solution favoured by hardware vendors is that they should provide a high performance code generator themselves. This approach is also favoured by the developers of TDF, since a single very good installer is needed per machine architecture.

39. The paper "Formal Program Developments" by Dr Cazin (FR) described the development language DEVA, a prototype language aimed at formally specifying a set of steps leading to a program. The language techniques for abstracting a formal development were illustrated. The developed objects would be known to be correct by construction; the method uses typing rules to check the validity of the developed object. Reuse of these highly complex specifications was suggested as a means of solving sets of similar problems.

40. In the discussion it was suggested that the problem of debugging a complex formal development, thus ensuring the correctness of the design rules that are employed, is comparable with the difficulty of debugging the resulting program. However, if a formal development is considered as a data structure, it can then be run through a proof checker, thus automating some of the burden of proof of the formal development.

41. Dr Jack Wileden (US) presented the paper "Automated Support for the Development and Evolution of Complex Software Systems" which reviewed some aspects of the ARCADIA research program, namely OROS and its object management capabilities, and tools for analysis. ARCADIA attempts to break down the distinction between programming language facilities, database facilities and filestore. There are three aspects to this ARCADIA approach: strong abstract typing, transparency of persistence and tool interoperability.

42. Two questions were asked concerning the manner in which OROS deals with garbage collection of persistent objects and concurrent access. Both questions were cited as difficult problems, currently being worked on, but needing more work.

43. A comparison of the PCTE and CAIS interface specifications with OROS was requested. Both PCTE and CAIS use a separate database management system, an Entity Relationship Attribute Database. The database functions are callable from languages used by PCTE/CAIS. These systems were observed to be based on 1970's technology - a useful development system for current use - whereas OROS and Ten15 were aimed at the next generation.

44. The question was raised and considerable discussion followed on whether money should be spent on an excellent implementation of PCTE or on research projects such as ARCADIA and Ten15. No consensus was obtained.

CHAPTER 5

5. C3I SYSTEMS DEVELOPMENT

Session Chairman: Dr J Grosche (Germany)
Rapporteur: Dr L Simcox (UK)

5.1 Introduction

45. This session considered the problems of C3I Systems development as a special case of military information system engineering. Four papers were presented ranging from the conceptual to the experimental.

5.2 Summary of presentations

46. Herr W Storz (GE) in his paper "A Structure for Distributed Command and Control Information Systems using Commercially Available Software" reported on an experimental implementation of a C3I system called EIGER. EIGER gives an example of a general architecture for a distributed command and control information system using:

- the ISO/OSI reference model
- the Ada language
- off-the-shelf commercial software (COTS)

47. The use of a suitable protocol suite based on the ISO/OSI reference model provides a functioning communication system and facilitates interoperability.

48. The layered model handles security by extra sub-layers (not defined in the basic ISO/OSI model).

49. The higher level of EIGER is completely written in Ada. To use COTS, in particular communication software and database systems like ORACLE, it is necessary to transform data structures from Ada to the C programming language and to define interfaces to control parallel transactions handled by the COTS database. EIGER was built as an experimental version to gain experience. More work is still needed to adapt and integrate available modules (COTS) with security components and specific applications such as expert system components.

50. A Bories (FR) in his paper "Object Oriented languages as the answer for interoperability and incremental development of Command and Control Information Systems" described the advantages of using Object Oriented

Languages (OOL) in CCIS. After an historical overview he gave a brief introduction into the nature of object oriented languages. Then he considered CCIS characteristics and the corresponding advantages of using OOL:

CCIS Characteristics	**OOL Help**
man-in-loop	rapid prototyping
interoperability	object description
evolution	granularity
security	
- confidentiality	- hidden data
- integrity	- inheritance
- reliability	- plausibility
- testability	

51. C3I systems need a special development methodology and object-oriented languages are a promising approach to control complexity.

52. P Y Simonot and S L Auboin (FR) presented "Army C3I system software design: a case study". P Y Simonot described the history of the activities on Army C3I systems. Two prototypes and a test-bed were developed beginning in 1968. This preparation allowed for extensive system modelling to define the architecture (hardware and software) of the system with respect to redundancy, performance, survivability and functionality. S L Auboin explained the results, ie the structure of the operational system. Three types of hardware components are used:

- database management computers
- user interface devices
- communication processors.

53. The software consists of three layers:

- basic standard packages (UNIX, GKS, X-WINDOWS, TCP/IP, CLIO database management system, etc)
- configuration dependent layer and system supervision
- application layer (configuration independent).

54. Specific features which were realised in the operational system were:

- survivability by hardware redundancy and software duplication
- adequate response time, by distributed database access
- on-line configuration management.

55. Concluding, P Y Simonot gave some research recommendations:

- evaluation methods and validation tools for checking survivability functions

- modelling of distributed systems for performance evaluation

- development methods.

56. C A DeFranco Jr (US) presented the paper "C2 for the 90's - new ideas in survivability". The main conclusion was that survivability should be achieved by "Replication of Function and Dispersion" rather than hardening, which is very costly and less effective. Consequently two technology issues are of importance: the distribution of data and survivable communication. He described three research and development experiments:

- Multi-media Communications Capability (M2C2), an in-house system to combine diverse physical transmission media to help obtain high levels of connectivity.

- Planning in a Distributed Computing Environment (PDCE), a system to evaluate and demonstrate reconnection strategies.

- The Survivable Adaptive Planning Experiment, aimed at demonstrating advanced technologies towards creating a deployable, survivable strategy planning system in a highly stressed environment.

5.3 Discussion

57. The main part of the discussion revolved round the role of the user during system development, which was agreed to be essential. The development method must be chosen accordingly, and prototyping is a promising way (the incremental development, evolutionary approach). However, "toy" prototypes are likely to be unacceptable to the military user. These users are conservative in terms of technology (reliability), so the quality of prototypes is essential for acceptance.

58. Another topic of discussion was the question of standardization of object oriented languages. C++ and Smalltalk seemed to be emerging de-facto standards. Performance is still a problem, but new, more powerful workstations would significantly improve the situation.

CHAPTER 6

6 CONCLUSIONS

6.1 Final Session

Session Chairman: ICA J P Crestin (France)
Rapporteur: P Y Simonot (DRS, Secretary Panel 11)

59. The objectives of the final session were to summarize the symposium results and to identify possible future research activities based on the input from the previous sessions. The session chairmen provided a short overview of the topics covered during their session: these proved to be significantly complementary.

60. The role of formal methods, which had been raised in several sessions, and their compatibility with life cycle models were discussed in depth and a number of issues were identified. Formal methods are supposed to capture the design process in a formal way. This implies that decisions made during that process (or at least the result of the decisions) should be represented in some way. The sensitivity of the outcome to the decision made is also an issue: can small "perturbations" in the assumptions change the final product significantly? Another possible issue is the relationship with prototyping. It was stated that formal methods related to prototyping by considering a restricted specification, but that there were no fundamental differences. Formal methods used together with a knowledge-based system provide an "animation" of the specified system. Formal methods might be used for critical components (such as security) when prototyping was not possible.

61. It was generally agreed that although Formal Methods are currently immature, they will become an important tool in the longer term and are a promising topic for future research. Other emerging technologies include prototyping, object-oriented programming, knowledge-based systems, persistent data systems and techniques for software portability. Many different and promising paradigms are being used. However, the integration of the different development paradigms can cause problems, particularly in the approach to system integration.

62. Further work is needed to define the role of prototyping and its relation to the system development model. An important element for study is the means of transition from prototype to engineered operational system. The infrastructure for operational systems seems less well developed than that

currently being applied within the prototyping environments. To ease the transition further work should identify improvements for the operational infrastructure and how they might be achieved.

63. The development of C3I systems requires specific approaches to solve problems such as the integration process, the use of commercially available hardware and/or software including Open Systems standards, the re-usability of software or more specific military issues such as interoperability and survivability. The system integration phase is probably the most critical phase in the C3I system life cycle and tools are needed to support it. The problems raised are both technical and procurement policy issues. From a technical point of view, this type of system is very complex and requires a large variety of engineering disciplines. On the other hand, taking into account that any software product has defects, the contractor liability for the system might be an issue. This is just one example of the wider issues raised by such system development models. Such issues were not considered as direct research problems.

64. The final session identified several areas to be considered for future activities of Panel 11, including Formal Methods, Software Re-usability, Portability and Interoperability, the use of Commercial Off-The-Shelf Software and Knowledge Engineering in C3I.

65. To summarize, a large variety of technologies were presented and discussed during the symposium. All of them could be used in order to solve complex problems in a very complex environment such as military information systems.

6.2 Conclusions

66. Several emerging technologies were identified as having potential benefit for future military systems. Various technical problem areas were agreed to need continuing research. In addition, the discussions identified significant problems concerned with the mapping of new system development methods and life-cycle models to procurement practices.

67. Military operations place increasing reliance on the use of software based systems. Systems in which software has a critical part range from embedded weapons systems to large distributed C3I. The characteristics required of military C3I systems include security, reliability, interoperability and survivability. This implies certain properties of the system architecture such as distribution and reconfigurability. System development is carried out against a background of rapidly developing commercial hardware technology offering considerable price and performance

benefits. However, commercial developments in systems engineering have not kept pace with this hardware technology improvement and do not adequately address specific military requirements.

68. Research and experimentation has shown the value of an incremental and evolutionary development process including prototypes. However, in considering the impact of emerging technology the problems are not entirely technical. Many are connected with the acquisition culture in defence which is strongly connected to a waterfall model of systems development. Current procurement practice was felt to legislate against the benefits expected from incremental and evolutionary development.

69. Thus, the symposium discussions found that it was difficult in the context of current procurement practice to evaluate the benefits offered by emerging technology. The involvement of the military sponsor and the future operational user from an early stage is essential in incremental and iterative development. No recognized mechanisms exist to ensure this involvement.

70. There is an urgent need for a transition strategy, including consideration of changed procurement practices, in order to allow full use of beneficial emerging technology in the engineering of future military information systems.

This page has been left blank intentionally

LIST OF PARTICIPANTS

J L Auboin	DGA/DAT/SEFT, Fort d'Issy, 92131 Issy-les-Moulineaux, France Tel: +33 4095 3485
Ing R Balducci	ITALTEL, Defence Telecommunication Division, via Tempesta 2, 20149, Milano Tel: +39 2 4388 2494 Fax: +39 2 4388 3270
Dr B M Barry	Defence Research Establishment, Ottawa, 3701 Carling Avenue, Ottawa, Canada K1A 0Z4 Tel: +1 613 998 2093 Fax: +1 613 990 8401
Miss S G Bond	RSRE, Superintendent Computing Division, St Andrews Road, Malvern, Worcs WR14 3PS Tel: +44 684 894997 Fax: +44 684 894303 Telex: 339747-8, Email: sgb@uk.mod.rsre
A Bories	ALCATEL ISR, 523 Terrasses de l'Agora, F-91034, EVRY CEDEX, France Tel: +33 1 6091 2275 Fax: +33 1 6091 2200 Telex: 600815F
A Bramley	UK(Air), Command Computer Officer, HQ Strike Command, RAF High Wycombe, Bucks
A J Burton	RSRE, CS2 Division, St Andrews Road, Malvern, Worcs WR14 3PS Tel: +44 684 895809 Fax: +44 684 894303 Telex: 339747-8
J Cazin	ONERA-CERT/DERI, 2 Av Edouard Belin, 31055 Toulouse Cedes, France Tel: +33 6155 7055 Fax: +33 6155 7112 Telex: ONECERT 521 5967 Email: cazin@tls-cs.cert.fr
ICA Chezlemas	DGA/DCAe/STTE, 129 Rue de la Convention, 75015 Paris, France Tel: +33 1 4425 8702 Fax: +33 1 4557 7620 Telex: STECTELECAERO
ICA J-P Crestin	STCAN, 8 Boulevard Victor, F75015 Paris, France Tel: +33 1 4059 1063 Fax: +33 1 4059 1932
D W Fikkert	TNO Physics and Electronics Laboratory, P O Box 96864, 2509 Den Haag, The Netherlands Tel: +31 70 3264 221 Fax: +31 70 3280 961 Email: dfikkert@ccintl.mod.uk (internet)
C A De Franco	RADC/COTD, Griffiss Air Force Base, New York 13441-5700, USA Tel: +1 315 330 2805 Fax: +1 314 330 3911
Dr K L Gardner	Head Defence Research Section, NATO HQ, B-1110 Brussels, Belgium Tel: +32 2 728 4420 Fax: +32 2 728 4103

K Geary	Sea Systems Controllerate, MOD, F Block, Foxhill, Bath BA1 5AB Tel: +44 225 883403 Fax: +44 225 882854
Dr W M Gentleman	National Research Council of Canada, Division of Electrical Engineering, Montreal Road, Building M50, Ottawa, Ontario, Canada K1A OR8 Tel: +1 613 993 3857 Fax: +1 613 952 7998 Email: gentleman@nrcdee.nrc.ca
G Giannino	NATO/NACISA, Head of Ada Support and Control Capability, 9 Rue de Geneve, B-1140 Brussels, Belgium Tel: +32 2 728 8388 Fax: +32 2 242 1022 Telex: 25931
Dr J Grosche	FGAN/FFM, Neuenahrer Strasse 20, D-5307 Wachtberge-Werthhoven, Germany Tel: +49 228 85 2288 Fax: +49 228 85 2451 Telex: +49 228 3647
Ir R de Haan	Physics and Electronics Laboratory TNO, Information Technology Division, P O Box 96864, 2509 JG, The Hague, The Netherlands Tel: +31 70 326 4221 Fax: +31 70 328 0961 Telex: 13185
Dr C Holland	Air Force Office of Scientifish Research, AFOSR/NM, Building 410 Bolling Air Force Base, Washington DC 20332-6448, USA Tel: +1 202 767 5025 Fax: +1 202 767 0466
F S Lamonica	USAF/RADC/COEE, Griffiss Air Force Base, New York 13441-5700, USA Tel: +1 315 330 2854 Fax: +1 315 330 3911
C Lewis	RARDE, CA1 Division, Fort Halstead, Sevenoaks, Kent TN14 7BP Tel: +44 959 3222 2508 Telex: +44 959 32971
M J Looney	ARE, AXC Division, Porsdown, Portsmouth, Hampshire PO6 4AA Tel: +44 705 21999 x2330 Fax: +44 705 21999 x3543
J Martel	RARDE, CA2 Division, Fort Halstead, Sevenoaks, Kent TN14 7BP Tel: +44 959 3222 x3956 Fax: +44 959 32971
or	DREV, 2459, Pie XI Boulevard North (P O Box 8800), Courcellette, Quebec, Canada GOAIRO Tel: +1 418 844 4698
R B G Mawby	NACISA, NATO HQ, UK National Experts Office, BFPO 49 Tel: +32 2 728 8248 Fax: +32 2 242 1022
Prof J McDermid	University of York, Heslington, York YO1 5DD Tel: +44 904 432782 Fax: +44 904 432767
Dr Morganti	ITALTEL, Central Research Laboratory, I-20019 Settimo Milanese, Italy Tel: +39 2 43887353 Fax: +39 2 4388 8462 Telex: +43 314840
Dr N E Peeling	RSRE, CS2 Division, St Andrews Road, Malvern, Worcs WR14 3PS Tel: +44 684 895314 Telex: 339747-8 Fax: +44 684 894303 Email: peeling%hermes. od.uk@relay.mod.uk

Dr R Pendeville Shape Technical Centre, P O Box 174, 2501 CD The Hague, The Netherlands
Tel: +31 70 314 2274

M Perre TNO Physics and Electronics Laboratory, P O Box 96864, 2509 JG, The Hague, The Netherlands
Tel: +31 703 26 4221 Fax: +31 703 28 0961

P R H Place Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213-3890, USA
Tel: +1 412 268 7746 Fax: +1 412 268 5758

Lt Col M Sciorella Italy MOD, SEGREDIFESA, via XX Settembre, 123/A, 00100 ROME
Tel: +39 6 4817805 Fax: +39 6 481 4264 Telex: DTMT-I 613436

Dr C T Sennett RSRE, CS1 Division, St Andrews Road, Malvern, Worcs WR14 3PS
Tel: +44 684 895184 Fax: +44 684 894540 Telex: 339747-8

Dr L N Simcox RSRE, AD4 Division, St Andrews Road, Malvern, Worcs WR14 3PS
Tel: +44 684 894693 Fax: +44 684 894540 Telex: 339747-8

Dr P Y Simonot Defence Research Section, NATO HQ, B-1110 Brussels, Belgium
Tel: +32 2 728 4759 Fax: +32 2 728 4103

I J Sinclair Real Time Engineering Limited, Capital House, 20 Park Circus, Glasgow G3 6BE
Tel: +44 41 322 9400 Fax: +44 41 331 1094

Dr D R Smith Kestrel Institute, 3260 Hillview Avenue, Palo Alto, California CA 94304, USA
Tel: +1 415 493 6871 Fax: +1 415 424 1807
Email: smith@kestrel.edu

Mrs M Stanley RSRE, CS2 Division, St Andrews Road, Malvern, Worcs WR14 3PS
Tel: +44 684 894576 Fax: +44 684 894303 Telex: 339757-8

Dr W Storz Forschungsinstitut fur Funk und Mathematic, Neuenahrerstrasse 20, D-5307 Wachtberg-Werthhoven, West Germany
Tel: +49 228 852 511 Fax: +49 228 852 451 Telex: +49 228 3647

V K Taylor DND/CRAD, Department of National Defence, Ottawa, Canada K1A 0K2, attn: DRDCS-2
Tel: +1 613 995 8008 Fax: +1 613 996 0038

ICA B Vors DGS/DRET, 26 Boulevard Victor, Paris, F00460 Armees, France
Tel: +33 1 4552 4666 Fax: +33 1 4552 4681 Telex: DTEPA 204648F

T A D White RSRE, St Andrews Road, Malvern, Worcs WR14 3PS
Tel: +44 684 894951 Fax: +44 684 894540 Telex: 339747-8

Dr J C Wileden University of Massachusetts, Computer and Information Science Department, Amherst, Massachusetts 01003, USA
Tel: +1 413 545 0289 Fax: +1 413 545 1249

This page has been left blank intentionally

TABLE OF CONTENTS

NATO UNCLASSIFIED

REPORT DOCUMENTATION PAGE

1. Recipient's Reference:	2. Further Reference:
3. Originator's Reference: AC/243(Panel 11)TP/1	4. Security Classification: UNCLASSIFIED/UNLIMITED 5. Date: 15.04.91 — 6. Total Pages: 23

7. Title (NU):

Management and Control of Prototyping as Part of the Development Life Cycle

8. Presented at:
AC/243(Panel 11) Symposium on Military Information Systems Engineering - RSRE, Malvern, UK - 8-10 May 1990

9. Author's/Editor's:
M. J. Looney - I.J. Sinclair

10. Author(s)/Editor(s) Address:

ARE-PN	Real Time Eng.
Portsmouth	Capital House
P06 4AA	20 Park Circus
United Kingdom	Glasgow
	United Kingdom

11. NATO Staff Point of Contact:
Defence Research Section
NATO Headquarters
B-1110 Brussels
Belgium
(Not a Distribution Centre)

13. Keywords/Descriptors:

SOFTWARE LIFE CYCLE - PROTOTYPING - MANAGEMENT AND CONTROL

14. Abstract:

This paper shows how the inclusion of a prototyping approach in the life cycle paradigm for software development of large complex systems need not result in an uncontrolled process. The procedures for using a prototype on which to carry out risk reduction can be specified and the necessary documentation generated. These stages have been identified and the structure of the documentation is indicated in relationship to existing development standards such as DoD-STD-2167A.

NATO UNCLASSIFIED

Management and Control of Prototyping

as part of the

Development Life Cycle

Authors.

M.J.Looney.
Head of Software Engineering Section (AXC4)
Command & Control Division
ARE-PN
Portsmouth PO6 4AA
Tel 0705 219999 (Ext 2330)
Fax 0705 219999 (Ext 3543)

I. J. Sinclair
Real Time Engineering Ltd
Capital House
20 Park Circus
Glasgow G3 6BE
Tel 041 332 9400
Fax 041 331 1094

Table of Contents

1.0 Introduction

The conventional 'waterfall' (Ref 1, 2) approach has been applied to large system development with a marked lack of success over the last decade or more. Over the last few years there have been several attempts to change the paradigm and to increase the emphasis on the 'front end' of the process, with the use of prototyping as a possible means of improving the resulting system. However there are several perceived shortcomings in the adoption of this approach the main one being the problems associated with the management and control aspects. Planning and control of prototyping is considered to be more difficult because the form of the evolving system, the changes which will occur, and the user requirements, can be largely unknown at the outset. This lack of explicit structure and form to the planning and control of the process is said to increase the difficulties of the management and can result in inadequate documentation and testing.

This paper does not seek to prescribe methods for prototyping or conventional development, but instead concentrates on themes which are central to the success of the marriage between them:
- how can information be transferred between the two types of process?
- can an approach be found which supports management visibility in a prototype without destroying the very nature of the process (i.e. experimentation at speed).
- when can prototyping be justified, and how?
- how can a project transition successfully from one phase to the other?

Having viewed the negative impacts which can result from management insistence on slavishly following standards in conventional development processes, it is hardly surprising that there is a natural reluctance within the prototyping lobby to see "standards" imposed on the prototyping phases. Clearly the application of a "waterfall"-type standard to a prototype is wholly inappropriate, and so other solutions must be sought.

The harsh reality is that prototypes must be commercially viable. They require to be justified, to be managed, and to achieve the prime objective of contributing to success in a cost-effective manner, by discovering and disseminating requirements and solutions which demonstrably save time and effort or improve functionality/performance in latter stages of development.

The paper was produced in the context of work being undertaken in looking at DoD-STD-2167A (Ref 3) and trying to identify the ways in which the standard could be made more palatable to those advocating rapid prototyping, while retaining the standard's ethos of a strong emphasis on visibility to permit management control.

2.0 Background

In the 1970's many attempts at producing complex real-time systems were unsuccessful. It was very difficult to point to significant success stories, while projects seemed abundant which were viewed by their instigators as basically disastrous in terms of cost and time scale overruns or in terms of unacceptable performance in functionality and/or responsiveness. Engineering managers of say, overall shipbuilding programmes, came to view the sophisticated computer systems elements of the programme as major risk areas and ones over which they could effect frustratingly little management control.

Lack of "visibility" was perceived as the major problem at that time and papers such as Ref.4 talked of the need to dispel the image of software as a "black art" and to enforce engineering disciplines on it which would generate the visibility required for management control. A strong emphasis was placed on the need to plan clearly the sequential stages which would lead to the final system and to generate documentary evidence at each stage, be it design document, test strategy or performance analysis, to permit assessment by a third party of whether or not progress was to plan.

Hence standards such as DoD-STD-21 67A evolved. The traditional "waterfall" model of the software life cycle lent itself extremely well to the identification of sequential stages and so was readily adopted. Documents required at each stage were defined in great detail, in the case of DoD-STD-2167A (through its supporting DIDs) to the level of detailing the exact format of the documents. Enforcement of such standards was seen as the panacea for the problems of the software industry.

However, in the 1980's, the continuing attempts at producing complex real-time systems do not seem to have created an improved success rate. Engineering managers who have enforced the standard now find significant documentary visibility being generated, but somehow the problems of cost escalation, [illegible] scale slippage and under-performance have not gone away. This may in part be due to the requirements and expectations of real-time systems, particularly in the area of responsiveness, growing dramatically in line with advances in hardware technology. It must, however, at least in part also be due to adherence to the standard being insufficient to solve the problems, or to the effect of invoking the standard not being as envisaged by its authors. It is the opinion of the authors of this report that both these conditions apply to a degree. The former in that controlled prototyping is a necessary adjunct, and the latter in that the standard's strong emphasis on document format rather than content has led to documents becoming an end in themselves, rather than a means to an end, in a way never envisaged by its authors.

3.0 Life cycles

In a document produced by ISO there is a list of activities under "reference model for software development" and within that list there are five distinct life cycles. The acceptance of the classical approach to the software life cycle represented by the internationally known 'waterfall' diagram, with its continual iteration and feed back from any level to any other has not been successful.

In the UK the DTI and the NCC produced the STARTS GUIDE (Ref 5) and introduced their version which bent the waterfall in the middle and was known as the "V diagram" .This also explicitly introduced the links between various decomposition stages with the testing and acceptance during the appropriate integration phase. However the basic problem with the waterfall diagram remained.

In the papers written in the early 80s, Balzer, Cheatham and Green (Ref 6) identified a new paradigm for the development of software. In this they identified that the basic approach adopted so far had been flawed in that there had been little or no computer support for the processes involved due to the relatively low cost of people against that of the expensive early computers.The strategy they put forward was that the development life cycle should be based on the use of automation to the highest possible level and that it should be split into two basic segments. The first representing the requirements analysis stage which encompassed the capturing of the requirement, the use of prototyping techniques to "validate" that the requirement was correct and that it represented what the end user had asked for. The output from this stage being a model of the requirement as a formal specification represented by the prototype. The second stage would be supported with tools capable of automatically generating a program to satisfy the formal specification. This would remove the manually element in the code generation and avoid the need for intermediate design, code and testing. All enhancements would be carried out at the first stage using the prototyping approach to identified the ramifications of the changes.

Alan Davis in his paper on comparing life cycle models (Ref 7) indicates that there is a long way to go to achieve the "ultimate" model that can identify the requirements before the user really knows what he wants, but, even so, some changes are needed. The adoption of a prototyping /evaluation /evolution approach does now appear to have a lot of support, including that of the Report of the Defense Science Board Task Force on Military Software (Ref 8).

The approach introduced by Barry Boehm (Ref 9) does seem to be moving in the this direction. In the approach adopted there has been a radical look at what is taking place and what the needs are to achieve an acceptable system,

using some of the results from work carried out on a rapid prototyping approach and comparisons with conventional development (Ref 10). In these, it was seen that there was a significant reduction in the time taken to produce a working system adopting the approach of continual modification of a prototype against the standard requirements specification/build approach. The use of continuous evaluation and risk analysis as an integral part of the process provides a much better basis on which to make the necessary decisions over options that may be open or which route may be most cost effective. When a prototype has been evolved to a satisfactory level and further refinement is considered not cost effective, then it is possible to engineer it to achieve the final product. The philosophy is simple, build a prototype, evaluate it, evolve it, when its good enough, engineer it.

The interesting aspect to this is that when the comparisons were carried out between the prototyping approach and the specification approach by several university groups, concern was expressed with the prototyping approach because they saw difficulties with the maintenance and enhancement of the system. The conclusion being that there was a need for further research in this area.

4.0 Prototyping

It is easy to sympathise with the programme manager who, having faced major problems with software in his projects in the 1970's, duly enforced DoD-STD-2167A to achieve better visibility on his project in the 1980's, then reads in the Report of the Defense Science Board Task Force on Military Software (Ref.8) that DoD-STD-2167A 'continues to enforce exactly the document-driven, specify-then-build approach that lies at the heart of so many DoD software problems'.

When he reads further in this report that the solutions to his problems are now seen as rapid prototyping involving a 'lash-up of handy components' he might be forgiven for despairing. How is he to control the progress of this 'lashing-up of handy components' and to monitor what it has achieved? Surely the standard was not wrong in enforcing visibility to permit management control?

The rationale behind any approach to introduce rapid prototyping to DoD-STD-2167A should be governed by the view that the basis for the standard is fundamentally sound in terms of the need for support in the procurement of large complex systems under contract. The standard has been widely used and feedback from this should be used constructively to improve it rather than destructively to discredit it. Having spent considerable effort (and money) on developing DoD-STD-21 67A the DoD may well lose considerable credibility if it now adopts the stance that the basis for the standard was wrong and that it is necessary to effectively discard it and start again. A more credible approach is to adopt the position that:

(a) The standard's basis of enforcing high levels of visibility to permit management control is fundamentally correct but the emphasis on mandatory documentation in mandatory format has been too extreme, leading to unintended results.

(b) Software is still evolving as an engineering discipline and the standard must evolve with it. The Defense Science Board Task Force report asserts that "We believe that users cannot, with any amount of effort and wisdom, accurately describe the operational requirements for a substantial software system without testing by real operators in an operational environment, and iteration on the specification. The systems built today are just too complex for the mind of man to foresee all the ramifications purely by the exercise of the analytic imagination". This assertion, and its implication that prototyping must become a prominent feature of complex system developments, is now gaining increasingly wide acceptance. Accordingly, DoD-STD-2167A must evolve to encompass this.

Point (a) above implies the need for clarification and some relaxations to the documentation requirements of the standard.

Point (b) above implies additions to the standard to cover the rapid,prototyping approach.

This work attempts to provide a suitable addition, but in such a fashion as to comply with the standard's ethos of a disciplined engineering approach yielding high levels of visibility to permit management control.

Before progressing further it is essential that several major issues associated with the use of a prototyping approach are clarified. The following paragraphs indicate the position adopted by the authors of this paper on these matters.

4.1 What is 'Rapid Prototyping'?

The first issue to be resolved is "What exactly is a prototype?". The answer is that the term appears to be used in a very wide-ranging sense varying from a very simple, cheap-to-produce system created for demonstration purposes (probably the most common interpretation) to an extensive, very formal system where the prototype in fact becomes the embodiment of the production system requirement as advocated for example in the forward-looking paper by Balzer, Cheatham and Green . The Defense Science Board Task Force report offers the following description and it is fair to say that any approach which broadly matches it is following a "rapid prototyping' approach:

"As people have recognised that the requirements, and especially the user interface, require iterative development, with interspersed testing by users, there has developed a technology for constructing "rapid" prototypes. Such a prototype typically executes the main-line function of its type, but not the countless exceptions that make programming costly. It usually does not have complete error-handling, restart or help facilities. The prototype is often built using a lash-up of handy components that swap performance for rapid interconnectability. It is usually run on a computer that is bigger and faster than the target machine."

This description is not unreasonable though its two final sentences are more appropriate to user-interface prototyping than performance prototyping which can be used as the basis for a more evolutionary approach.

4.2 Why Prototype?

It is very important before embarking on any prototyping exercise to clearly establish the purpose of the exercise. Prototyping must not become something that is mandated to be done because it is required by a standard but rather something that is done because good engineering judgement suggests it is likely to be beneficial to the project in terms of those criteria against which the success of the project will be measured. This usually means some combination of reducing cost, time scale and/or risk.

Prototyping must always be associated with resolving some uncertainty and will usually be associated with either clarifying user requirements or establishing feasibility either technical or financial and thereby reduce the risk. Prototyping will cost money and take time, so careful consideration will require to be given as to its justification. Usually systems for which it will be effective are systems which are breaking new ground either in terms of user requirement or new technology (if a previous similar system is in existence cannot the lessons that would be learned from a prototype be learned instead from it?).

4.3 Identification of Derived Requirements

A very strong justification for the use of prototyping is to identify "Derived Requirements" in very complex systems. These are requirements which are initially invisible to the end-user when specifying the system but are a function of the way the user's specified requirements are implemented.
For example, consider the case of a ship-wide command and control system for which certain levels of reliability and vulnerability require to be met as well as defined rates of system responsiveness. If, to satisfy

the system's requirements for reliability and vulnerability, the designers opt for a distributed system with redundant nodes, then performance requirements for the underlying network and distributed data base become requirements derived from the initial system responsiveness requirements. If these derived requirements are not satisfied the system will not perform, even though these requirements were not directly visible when the system was originally specified. The feasibility of satisfying these derived requirements therefore becomes instrumental in establishing the feasibility of satisfying the original requirements. This is very much the domain of prototyping: "learning about the hidden requirements".

4.4 Feasibility of Non-functional Requirements

Reliability, availability, maintainability are examples of what can be termed 'non-functional' requirements. They are requirements which can be stated extremely simply, maybe in a single sentence, by end-users perhaps unaware of the enormous impact they can have on the technical or financial feasibility of implementing a compliant system. For example 'the system will have 99.9% availability' is extremely easy to write, but reducing the figure by a very small amount could completely change the technical/ financial feasibility of an otherwise identical project. These requirements are often the hidden factors behind development problems, as to retrofit this type of requirement actually means start again, they can not be added once development is underway

A paper-based feasibility study may give satisfactory answers to such questions prior to embarking on a major programme but often prototyping will be necessary to establish with confidence whether the technology exists, and a system design can be produced, to satisfy the non-functional requirements. In this type of situation it may be cost effective to 'save' the prototype and build on it rather than discarding it

4.5 Extent of Prototyping

In a complex system it is likely that uncertainty will only exist concerning certain aspects of the system. Again prototyping should not be done simply because it is mandated by a standard. Engineering judgement must be used to localise prototyping activities to those areas where the information gained through the exercise is likely to prove most beneficial to the project and be an effective use of resources.

Examples of two areas of systems in which prototyping is often an effective use of resources are:

(a) in clarifying the user interface.
(b) in establishing an underlying system infrastructure.

Unsatisfactory user interfaces and unsatisfactory response times are perhaps the two commonest user complaints about complex real-time systems.

The authors of this report subscribe to the view advanced by the US Defense Science Board Task Force that in a complex system a satisfactory user interface cannot be established without testing by real operators in an operational environment. No amount of imagination will substitute for actual feedback from end-users evaluating a prototype in the process of defining what a user interface should, and should not, do.

The authors also believe that solving the problems of how the underlying system infrastructure (for example, the distributed database and communications network in the Ship Command and Control System discussed earlier) is going to satisfy the systems responsiveness requirements, is such a central issue to so many of today's complex real time systems that this will be a clear candidate for prototyping activity in most large systems.

4.6 Procurement Issues

It is interesting to note that a recommendation of the U.S. Defense Science Board Task Force report is: "For major new software builds, we recommend that competitive level-of-effort contracts be routinely let for determining specifications and preparing an early prototype".

If this recommendation is implemented, then some reasonably formal framework for conducting the prototyping activity, as advocated here, will be of assistance in enabling the results of competing contractors to be compared.

4.7 Quality of Prototypes

A fundamental issue which must be addressed in any prototyping exercise is the planned life span of the prototype. Is it a quick lash-up, which will be discarded as soon as the desired lessons have been learned from it? Or, is it a very useful embodiment of the requirement which is to be used as a means of evolving into the production system?

There is no simple rule as to what quality of software and documentation should be produced for a prototype, but asking questions such as the ones above should enable the decision maker to make a practical judgement. In the extreme case of say a prototype for a major weapons system which would have serious consequences should it fail to perform even to a small degree, we would advocate that the prototype itself be developed to standards at least as stringent as DoD-STD-2167A. At the other extreme, where a relatively simple system user interface has been 'lashed up' to assist in its definition, with the full intent of discarding the prototype once it has served its purpose, then it would be acceptable to apply minimal quality standards for software and documentation to the prototype development.

The quality of software and documentation required for a prototype must therefore relate to its purpose, particularly in terms of life span and use.

4.8 When and How to Stop Prototyping

Prototyping is normally an iterative process, and as with most iterative processes, it is only useful if the process converges to some acceptable end-point. Prototyping is learning about the unknown and can be very interesting leading to the danger of more iterations being carried out than are actually beneficial to the project. The best guidance that can be given is that after each iteration, the same criteria which were originally used to decide whether or not to prototype should be used to decide whether another iteration is justified. This reassessment will need to take account of the amount of available resources (both financial and time scale related) which have already been committed to the prototyping activity.

4.9 Advantages and Disadvantages of Prototyping

To conclude this section on "major issues" the following is a summary of advantages claimed for prototyping tempered by some disadvantages which are likely to be encountered when the sizable step is taken from the philosophy of reports such as this to the harsh realities of established procurement paths in the real world.

4.9.1 Advantages claimed for prototyping

4.9.1.1 It saves money in the long term

The money invested up-front in the prototyping activity is likely to be repaid many times over by savings through not having to do major rework on the production systems to satisfy the end-user.

4.9.1.2 It reduces risk

If some aspect of the system is not feasible either technically or financially then prototyping can identify this early in the project thereby avoiding the risk of unnecessary major expenditure.

4.9.1.3 It leads to a better system

The system is likely to be better in the sense that end-users are more likely to be satisfied, particularly with the user-interface aspects.

4.9.1.4 It provides early visibility which is psychologically advantageous

This is an aspect of prototyping which should not be under played. The encouragement is considerable, to both end-client and the team developing the system, when some version of the system they are trying to create, albeit in prototype form, becomes visible and tangible.

4.9.2 Disadvantages of prototyping which have been perceived

4.9.2.1 It costs more in the short term

The harsh reality is that this will often prevent prototyping taking place. In a world where procurement processes are dominated by annual budget constraints a project which seeks significant up-front expenditure (to save money 3 or 4 years later) risks cancellation whilst one seeking smaller amounts of funding based on a view through "rose-tinted spectacles" of the position in 3 or 4 years time is much more likely to be approved.

4.9.2.2 It risks suffering the Prototype Paradox

If a prototype is not perceived to be very good, i.e., not particularly representative of the final system, then the client will say "What a waste of money. Let's not do it again". If a prototype is perceived to be good, then the client may say "Excellent. We'll have one of those. How fortunate we won't have to spend any more money".

The Prototype Paradox is most easily overcome with an 'educated' client who has seen the problems before and is aware of the major issues surrounding the use of prototyping.

5.0 The Prototyping Life-Cycle

A proposed Prototyping Life-Cycle model has been produced with a view to integration with DoD-STD-2167A, and those familiar with that standard will recognise some intentional similarities in nomenclature.

There have been many benefits seen in the use of KBS type development facilities with their closely knitted sets of tools allowing the generation of rapid prototypes which can be tried and assessed and it is considered essentially that this ability to experiment in order to identify the risk associated with various options open at the start of a large software based project become part of the accepted life cycle. A major problem with adopting this approach is the lack of management control and, as indicated earlier, the limited prospects for maintenance and enhancement which result from the lack of detailed information at the end of the prototype evaluation on what has been done and why.

Our approach uses the basic structure indicated in the spiral model, it includes the use of a prototype followed by evaluation and risk analysis. It then repeats the process evolving the prototype until an acceptable level of risk is identified in the evaluation and the product can be engineered from the information so far obtained. The whole process may initially run as a set of parallel prototypes which are examining different aspects of the system and these can be brought together on successive iterations. The approach also identifies the documents which would be needed at the appropriate points as part of the management/control process necessary for the contractual procurement of a large system and to allow for the through life maintenance and enhancement of any large system.

In creating this model and writing its description in a format compatible with DoD-STD-2167A the authors are conscious that their work may face the criticism (as DoD-STD-2167A has) of being too strongly "document driven". The authors wish to stress that the grand titles given to the proposed documents, their acronyms and their format are entirely secondary to their content and ability to satisfy their purpose of describing and recording useful information about what has been done or will be done. If certain documents were to be retitled or perhaps combined into one for ease of use on a particular project this would cause the authors no concern whatever. The reader is asked to bear in mind this idea of documents as tools to be used as and when appropriate, rather than driving forces, in reading the description which follows. It should be stressed that the documentation standards adopted for any project should not be the driving factor in the procurement, a project must tailor any standard to meet its own needs.

If we take the final quadrant of the spiral model to be the equivalent of the "waterfall" model which may be initiated at any convenient point in the process, then the existing requirements for management and control are already laid down for interpretation in 2167A. As this identifies the System Requirements Specification as the initial document from which the waterfall flows. The equivalent document in the new schema is the Initial System Requirements Specification (ISRS) which is the input to a decision making process whether or not to prototype (Prototype Decision Process) which will assess the options open at the outset in terms of the alternatives, the risk, the constraints and the objectives. If the decision is not to prototype the ISRS becomes the SRS and DoD-STD-2167A becomes applicable as normal. If the decision is to prototype then a spiral is entered which may be circumnavigated several times before proceeding to the waterfall!

6.0 Processes and Documents

This shows the processes which require to be carried out and the documents which require to be produced during prototyping activity. The stages are as follows.

6.1 Initial System Requirements Specification.

The Initial System Requirements Specification document (ISRS) corresponds to the System Requirements Specification (SRS).

6.2 Prototype Decision Process.

The contractor shall address whether or not prototyping should be applied to the system as defined in the ISRS. This decision shall be based on the level of risk associated with the system, based on such factors as: complexity of the system, the extent to which it is breaking new ground with regard to functionality and use of technology, the level of clarity of definition in the ISRS of areas such as the User Interface, and the constraints which must cover aspects of cost, and time scale. If the decision reached is that prototyping is not appropriate then the reasons for this decision shall be documented, the ISRS shall become the SRS and development shall proceed as per the accepted model. If the decision is that prototyping is appropriate then the following shall be produced/carried out.

6.3 Prototype Strategy Document.

The PSD shall contain at least the following: the reasons why prototyping is being undertaken and the purpose of the prototype; the extent of the prototype (i.e., which areas of the full system are being prototyped); the expected life span of the prototype and the quality levels to which prototype software and documentation shall be produced and maintained. These standards will determine whether this is a major formal process, possibly itself following the traditional waterfall life-cycle, or a quick "lash-up" the details of which will soon be unimportant.

6.4 Prototype Implementation Plan.

The PIP shall contain at least the following: a definition of the prototyping activities to be carried out; a schedule of events and allocation of resources for the work being undertaken.

6.5 Prototype Evaluation Description and Plan.

The contractor shall produce one or more Prototype Evaluation Description documents which will outline activities to be performed in evaluating the prototype and a Plan which will allocate responsibilities for conducting various facets of the evaluation.

6.6 Prototype Construction Process.

The contractor shall carry out the construction of the prototype in accordance with the Prototype Implementation Plan and to the standards defined by the Prototype Strategy Document.

6.7 Prototype Evaluation Process.

Evaluation of the prototype shall be carried out guided by the Prototype Evaluation Descriptions and the results of the evaluation recorded in one or more Prototype Evaluation Reports (PER's).

(This is the most important stage on the prototyping life cycle, the stage at which the important lessons will be learned. The evaluation will normally be carried out by end-users of the system if functional aspects such as the user-interface are under examination. Some aspects of the evaluation may be carried out by systems specialists, such as monitoring the traffic on a communications line to try to understand the likely performance under worst case conditions or to experiment with different configurations.
The Prototype Evaluation Description(s) and Plan will have outlined the various evaluation exercises and responsibilities for carrying them out. The term "outlined" is used to emphasise that a certain amount of freedom must be given to the evaluator to use his judgement in what is essentially a learning exercise. The Prototype Evaluation Descriptions must be viewed as tools intended to steer and focus the evaluation rather than as detailed instructions to be followed to the letter.)

6.8 Prototype Evaluation Reports.

The contractor shall produce one or more Prototype Evaluation Reports which shall encapsulate and communicate, in terminology understandable to both system user and system implementor, the deficiencies and strengths of the prototype and recommendation for changes, additions or clarifications to the system requirements.

(A crucial aspect of the evaluation process is that the lessons learned must be communicated to those who can apply the knowledge to the benefit of the production system. There is much to be said for joint participation in evaluation activities by system end-users and system implementors. Human intercommunication is such that more will be learned by implementors in terms of "understanding the system" through dialogue with end-users in a "hands-on" situation, than through written communication via some form of evaluation reports. Nevertheless, it remains necessary to produce such reports to preserve as well as possible the information gained for the future benefit of non-participants in the prototyping work.)

6.9 Requirements Revision Process.

On completion of the Prototype Evaluation Reports the contractor shall assimilate these reports and revise as necessary the ISRS (or RSRS if more than one prototype cycle has occurred) to reflect the new information which has been learned. This revision will create a Revised System Requirements Specification (RSRS).

(Some statements made will be precise and quantifiable and able to be fed directly back into a revised System Requirements Specification, e.g., "This display is confusing because X is adjacent to Y. It would be much improved by reorganising the information displayed like this....". Other statements will be much less tangible, e.g., "the system is difficult to use because..." hopefully dialogue between implementor and user will enable the implementor to write down more clearly the ways in which the user believes the system can be improved.)

6.10 Revised System Requirements Specification.

The RSRS may become the SRS or be further revised dependent upon the outcome of the Strategy Reassessment Process.

6.11 Strategy Reassessment Process.

On completion of a cycle of prototyping the contractor shall reassess the strategy for prototyping and decide whether further prototyping is justified using the same criteria as used for the original Prototype Decision Process. If the risk level is low enough then no further prototyping need take place and the RSRS shall

become the SRS and development shall proceed . If further prototyping is to take place then the PSD and PIP shall be revised accordingly prior to the next phase of prototyping.

6.12 System Requirement Specification.

The output of the prototyping activity shall be an SRS and supporting PER's which give sufficient definition and information about the System to let development proceed, as per the traditional "waterfall" life-cycle model with a minimum of need for iteration.

7.0 Conclusion

In moving on from the "waterfall" life cycle model, which has been shown not to be ideal for the procurement of large systems, to a paradigm which will carry forward some of the features which have been accepted as essential, such as the need for control and the ability to measure or evaluate progress against the objectives identifies at the outset. The premiss has been to evolve and adapt rather than attempting to propose revolution. While in some ways it can be considered as being a radical change to the old method in its move from a clear cut beginning with a statement of requirements which could be taken and "manipulated" to produce a "satisfactory" system for the end user, to one which emphasises that the start is very much a "grey" specification and that it requires considerable clarification; there is still a need for certain information which must be available. This information is necessary in any large project if management are to carry out their task successfully, even more so when contract conditions must be meet.

In the paper a structure has been imposed on the use of a prototyping approach and a set of documents and processes identified which will provide the information needed, based, as closely as possible, on DoD-STD-2167A. A similar nomenclature has been used for documents (e.g. a correlation will be noted between the software testing activities of 2167A and the prototype evaluation activities) to retain the standard's strong emphasis on visibility. From the outline given it is possible to see how the use of a rapid prototyping approach can be integrated with a conventional 'production' and still allow the retention of full control and management capabilities essential for the procurement of large complex systems with extended in service life expectancy.

The identification of the necessary documentation and the use of a prototype to investigate the "unknowns" thereby reducing the risk, will make the use of "waterfall" model a relatively straight forward process as most of the risk will have been removed and the implementation will at last become a "production" and not a lengthy development with all its associated problems.

8.0 References:

Ref 1. Royce, W.W. 'Managing the development of large software systems: Concepts and Techniques' Proc. WESCON. Aug 70

Ref 2. Boehm, B.W. 'Software Engineering' IEEE Trans Comput. C-25 1226-1241 Dec 76

Ref 3. 'Military Standard, Defense System Software Development.' US DoD Washington. DoD-STD-2167A 29 Feb. 88

Ref 4. Sinclair, I.J. 'The management of software with along life span' Proceedings of the Seventh Ship Control Systems Symposium (vol 4) UK 1984

Ref 5. 'The STARTS Guide' UK Department of Trade and Industry 1987

Ref 6. Balzer,Cheatham, and Green, 'Software technology in the 1990's: Using a New Paradigm' IEEE Trans. Software Engineering Nov.83

Ref 7. Davis, A.M.,Bersoff, E.H.,Comer, E.R. 'A Strategy for Comparing Alternative Life Cycle Models'. IEEE Trans. on Software Engineering Vol 14 No. 10 October 88

Ref 8. Report of the Defense Science Board Task Force on Military Software. July 1987 Defense Science Board Task Force Chairman Frederick P. Brooks, Jr.

Ref 9. Boehm, B.W. 'The Spiral Model of Software Development and Enhancement' ACM SIGSOFT Software Eng. Notes. 11(4):14-24 Aug 86 & Computer 21(5): 61-72 May 88.

Ref 10. Boehm, Grey, and Seewaldt, 'Prototyping vs Specifying: a Multi-Project Experiment' IEEE Trans. on Software Engineering 84

NATO Symposium

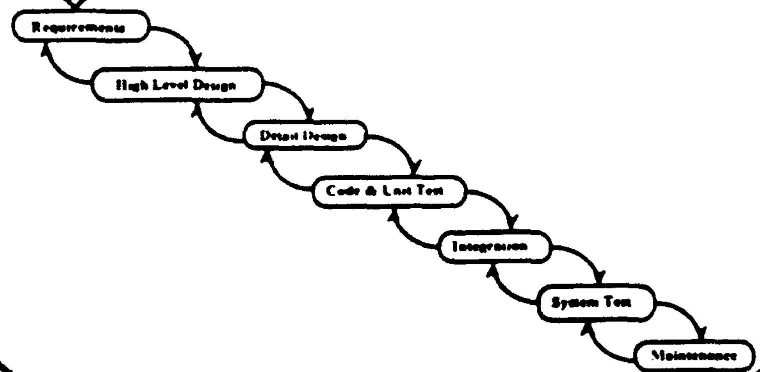
Management and Control of Prototyping

as part of the

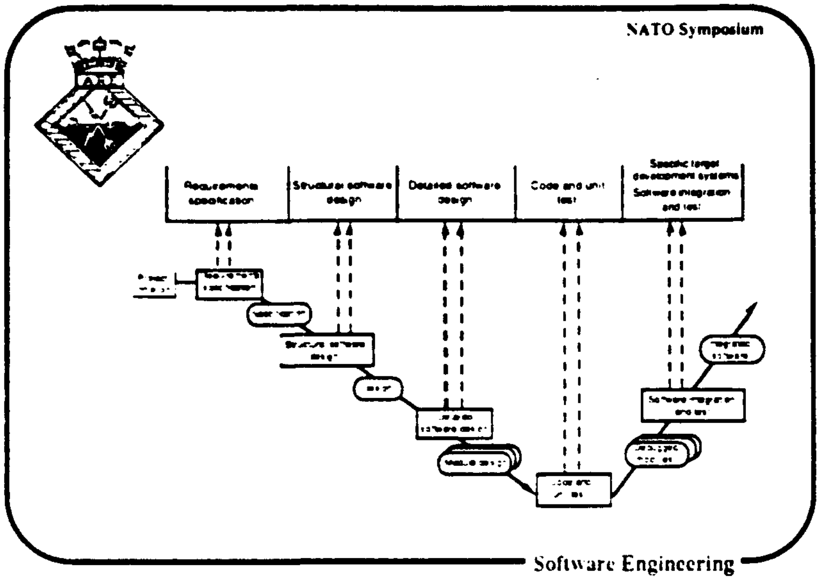
Development Life Cycle

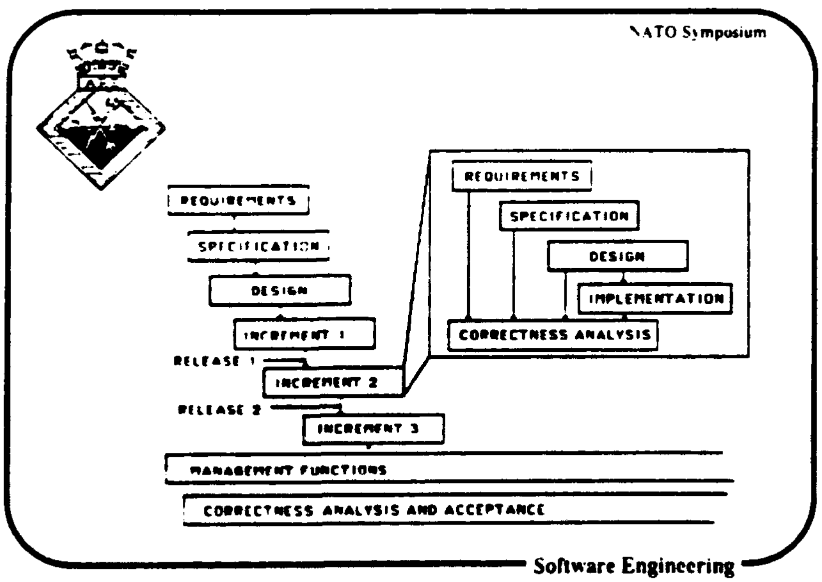
Software Engineering

NATO Symposium



Software Engineering


NATO Symposium
Requirements specification
Structural software design
Detailed software design
Code and unit test
Specific target development systems
Software integration and test
Software Engineering


NATO Symposium
REQUIREMENTS
SPECIFICATION
DESIGN
INCREMENT 1
RELEASE 1
INCREMENT 2
RELEASE 2
INCREMENT 3
REQUIREMENTS
SPECIFICATION
DESIGN
IMPLEMENTATION
CORRECTNESS ANALYSIS
MANAGEMENT FUNCTIONS
CORRECTNESS ANALYSIS AND ACCEPTANCE
Software Engineering

NATO Symposium

Software Life Cycle
(State of the Practice)

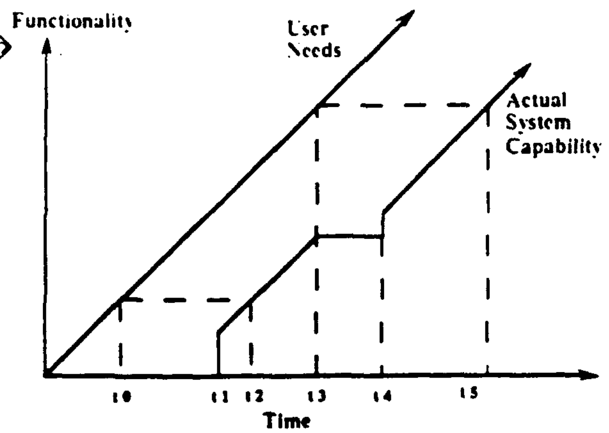
Missed schedules

Cost Estimates Exceeded

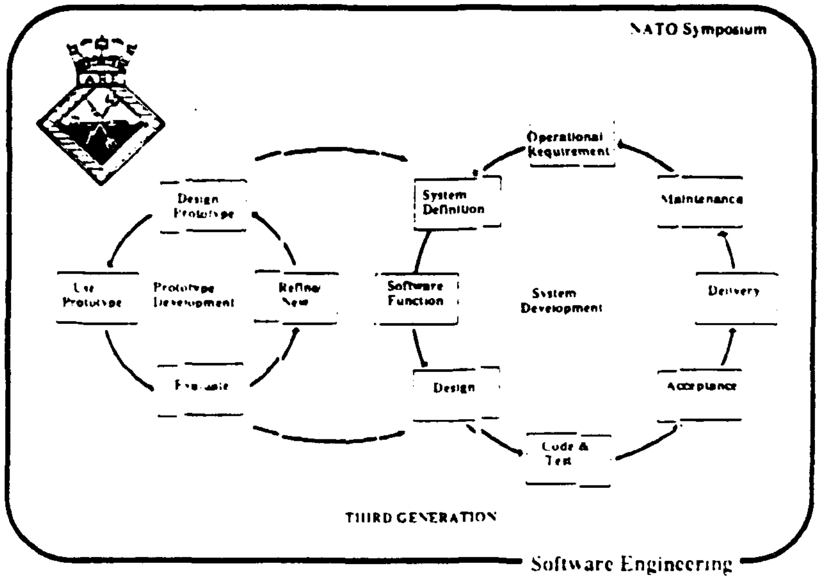
Little Use of Available Technology

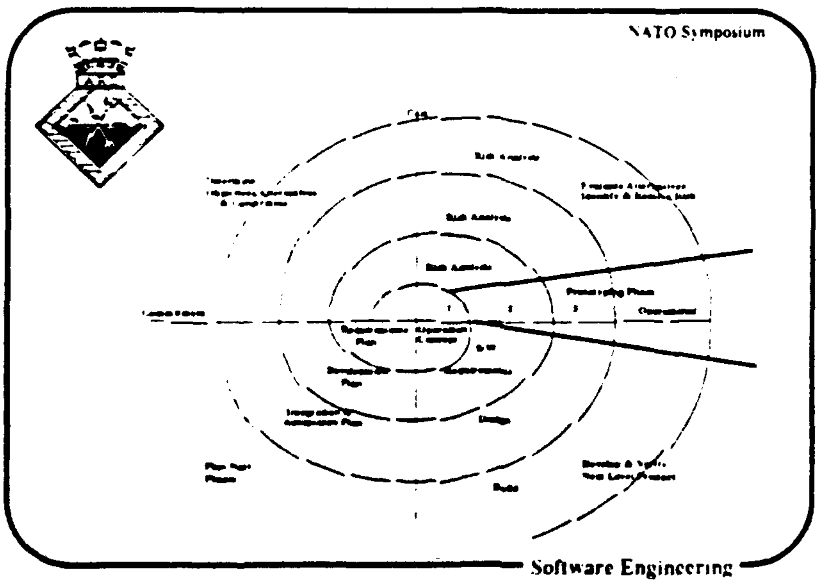
Software Engineering

NATO Symposium

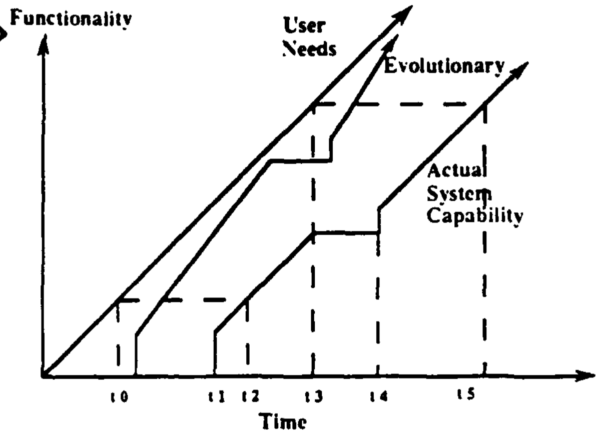


Software Engineering


NATO Symposium
Operational Requirement
System Definition
Maintenance
Design Prototype
Use Prototype
Prototype Development
Software Function
System Development
Delivery
Design
Acceptance
Code & Test
THIRD GENERATION
Software Engineering


NATO Symposium
Software Engineering

NATO Symposium



Software Engineering

NATO Symposium

UK
Chief Scientic Advisors Report
on
Software Intensive Projects

US
Report of the Defense Science
Board Task Force
on Military Software

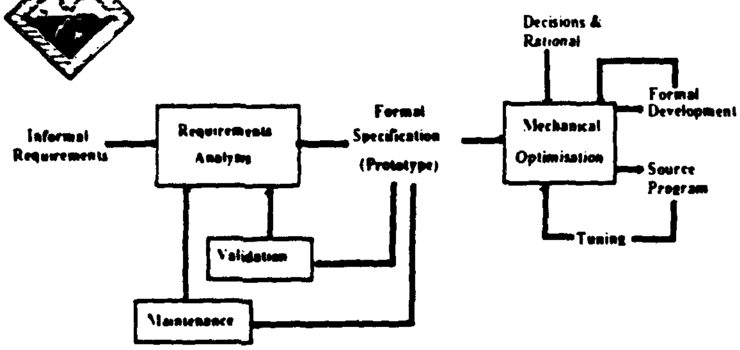
Software Engineering

NATO Symposium

" ...the importance of iterative development of requirements, testing against real user's needs by prototyping."

Software Engineering

NATO Symposium



NEW MODEL

Software Engineering

NATO Symposium

What is Prototyping ?

Why Prototype ?

Derived and Non-functional Requirements

Extent of Prototype

Procurement Issues

Software Engineering

NATO Symposium

A prototype typically executes the main-line functions but not the exceptions. It usually does not have complete error-handling, restart or help facilities

Software Engineering

NATO Symposium

Advantages of Prototyping:-

- Saving through Life Costs
- Reduces Risk
- Early Visibility
- More Acceptable

Software Engineering

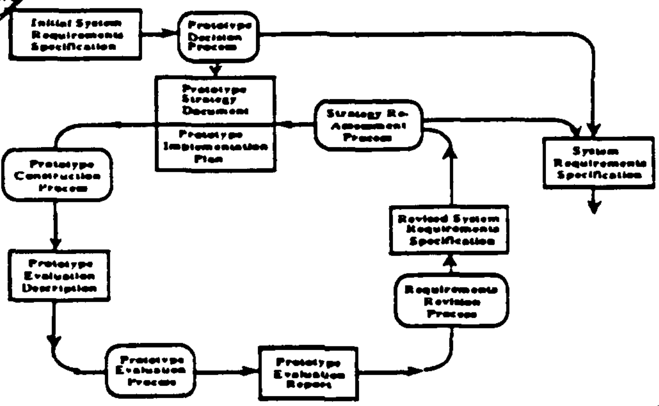
NATO Symposium

Disadvantages of Prototyping:-

- Short Term Costs
- Prototype Paradox

Software Engineering

NATO Symposium



Software Engineering

NATO Symposium

Conclusion:

The use of ManagedProtytping is an

Essential Part of System Development

and must be Included in the Life Cycle

Software Engineering

REPORT DOCUMENTATION PAGE

1. Recipient's Reference:	2. Further Reference:
3. Originator's Reference: AC/243(Panel 11)TP/1	4. Security Classification: UNCLASSIFIED/UNLIMITED 5. Date: 15.04.91 — 6. Total Pages: 6

7. Title (NU):

Engineering the Process of Software Evolution

8. Presented at:
AC/243(Panel 11) Symposium on Military Information Systems Engineering - RSRE, Malvern, UK - 8-10 May 1990

9. Author's/Editor's:
W. Morven Gentleman

10. Author(s)/Editor(s) Address:
National Research Council
of Canada
Ottawa K1A 0R6, Ontario
Canada

11. NATO Staff Point of Contact:
Defence Research Section
NATO Headquarters
B-1110 Brussels
Belgium
(Not a Distribution Centre)

13. Keywords/Descriptors:

EVOLUTION, REUSE, SOFTWARE COMPONENTS

14. Abstract:

Procurement and contract monitoring procedures focus on stages of software development prior to first delivery of the product, consequently software engineering has also. This is inappropriate for products with long life-times, where the requirements change over that life-time and the product must evolve to meet the changed requirements.

Cost, resource availability, delivery schedules, and reliability through exposure dictate substantial reuse of parts of earlier implementations. This also accommodates user experience with earlier implementations. Data produced for and by earlier implementations must be accessible or convertible. The transition must be planned and supported.

This paper discusses problems that must be addressed in evolving software, and tools and techniques that are available.

Engineering the Process of Software Evolution

W. Morven Gentleman*

Table of Contents

* Principal Research Officer
National Research Council of Canada
Ottawa, Canada

Engineering the Process of Software Evolution

W. Morven Gentleman
National Research Council of Canada
Ottawa, Canada

1. Evolution

The traditional view of the software life cycle is something like: a requirement is formulated, the specification formalised, an implementation designed, the design implemented, the implementation tested and validated, and then the software is delivered and enters maintenance mode. This view produces a focus on the early stages of the process, requirements through first delivery, and that focus is reflected in procurement procedures and contract monitoring. This in turn has fostered a focus of software engineering, and especially CASE (computer assisted software engineering) tools, on these early stages in the software development process. While there can be no dispute that the early stages of design are important, this concentration on the... unfortunate. Even in the traditional view, it is widely recognised that the majority of the cost, perhaps 70% to 80%, is after first delivery, and is only indirectly affected by improvements in the design stage.

The traditional view is inappropriate for a lot of software because it assumes that the requirements are fixed. There are many examples of software for which the requirements themselves change over time, and the software must evolve to meet these new requirements. Commercial shrink-wrap software, for example, has to issue significant updates at intervals of 6 to 24 months, not just to remain competitive by matching and bettering the features of their competitors, but because the customers will abandon a product if they don't believe the supplier is alive and well and actively improving it. An example related to military software is that embedded systems often have a lifetime measured in decades, and over that time the mission of the system may change drastically, and the operating environment for the system will change. The point in these examples is that the initial functional requirements may have little resemblance to the functional requirements needed at later times, and that technology to establish conformance to the traditional form of initial specification does little to assist in migrating the product to meet its new needs. Indeed, too narrowly meeting the initial specification can complicate evolution.

We stress that evolution is qualitatively different from maintenance, although they are often confused when looking at the downstream costs of software. Maintenance is generally understood to mean fixing bugs, making minor changes to improve compatibility with the environment (accommodating to new operating system releases, for instance), and adding minor enhancements that have little effect on the software as a whole. Evolution to meet new requirements can involve radical changes, and unfortunately sometimes is treated by completely rebuilding the software as a new project. We argue that such a response is rarely desirable. Cost and time to delivery are obvious detractions, but there are others. For instance, a completely independent implementation is likely to feel different in aspects not covered by the specification, and this can introduce retraining costs for experienced users. More importantly, we argue that the response of completely rebuilding the software as a new project is unnecessary, if thought had been given beforehand to the possibility, or rather certainty, that the requirements will change.

The key concept is the decision that because we know the requirements will change, the software should be built not so that it can meet the initial requirements, but that it can quickly and cheaply be made to meet any set of requirements in some range. It is useful to look at analogies outside the software domain. The crescent wrench has an advantage over a simple open end wrench in that it can be used when the size of nut is not known initially, and might even turn out not to be one of the expected standard sizes. The extension ladder and the Bailey bridge similarly solve a range of problems where a tightly specified device might solve just one. The 1/4 inch electric hand drill, through adapters, can be used in applications from sanding to sawing that need not have been considered at initial purchase. On a larger scale, flexible manufacturing in the factory, based on use of assembly and other robots, not only can economically provide the advantages of automation for much smaller production runs than hard automation would require, but facilitates quickly modifying the production line to produce related items in response to market demand. All these analogies carry the message that there can be a penalty for flexibility, in that the dedicated item may do its one job better or cheaper if that is the only job to be done, but that penalty is immaterial if the job changes. The same situation can obviously occur in software too, but, curiously, sometimes consideration of how to cope with a range of requirements can suggest generalisations that actually solve specific cases better than would the obvious approach for that case.

2. Reuse

A popular topic in software engineering is reuse, i.e. how to amortise development costs over a bigger base by using existing code rather than developing something afresh for each new product. Viewing the sequence of products resulting from evolution of requirements in this light should be rewarding, for surely successive releases of essentially the same program should have amongst the best opportunities for reuse.

The objectives of reusable software are not just reduced cost and earlier delivery. Consistency is another benefit, appreciated by users and maintainers alike — and by management who do not have to pay retraining bills. Improved reliability is a less obvious benefit, but using components which have had extensive field exposure can be one of the most effective ways of avoiding surprises from quirks and anomalies, never mind design errors. (The reliability benefits of off-the-shelf components versus custom design are well recognised for hardware.) The ability to do large scale prototyping is another objective of reuse. Prototyping many systems would not be practical if it were not possible to exploit the preexistence of subsystems, although those subsystems might not be exactly what would be wanted beyond the prototype stage.

The topic of reuse is an interesting one, because it does not arise naturally in all paradigms of the software development task. The most common paradigm discussed in the literature is the development of an isolated program, and in particular the development of a well understood, fixed function, production program. This is the paradigm for which the waterfall model of software development is plausible, i.e. the monotonic progression through the stages mentioned before from requirements analysis through independent verification and validation. It is well understood, however, that this paradigm is not universal, so another paradigm is often discussed, which is also the development of an isolated program, but in this case one for which the desired functionality is not well understood, indeed what might be a feasibility experiment. In this case the spiral model of software development is often advocated, in the belief that there really is some well-defined requirement and corresponding formal specification, but that it is just not known initially. By iterating through drafts of the requirements and specification, sometimes just through to preliminary designs, sometimes through to simulations, sometimes

through to prototype construction, and sometimes as far as through to field trials on real users, those hidden requirements are revealed, and a product results conforming to those requirements, much as with the waterfall model.

Reuse becomes apparent as an important topic when the paradigm of software development goes beyond the development of an isolated program, and one considers the development of related programs. Two important, and not mutually exclusive, cases exist. The first case is where there are families of programs that exist simultaneously, doing related but different tasks. Sharing components is essential, for reasons from limiting development and maintenance costs through to ensuring interworking by common interfaces. The design of these shared components must precede the design of the individual members of the family, each of which will depend on many components. This is necessary to ensure that the components are appropriate for all family members, but designing the components first of course inverts the approach of top down design for the individual family members. In practice what usually happens is that the initial design of the shared components turns out to be inappropriate for some family members, and the most desirable resolution is to revise the design of the shared components to fit all family members, and retrofit these revised components even into the family members that worked before, in order to achieve parsimony in the number of components that must be supported.

The other case of related programs is the one we are considering, that of a sequence of programs evolved from each other. What distinguishes this case from the family of programs produced simultaneously is that future requirements are unknown. However, the probable kinds of changes in requirements are often predictable, and so the difference between the two cases is more apparent than real.

We have chosen to use the term software components, rather than just software reuse, because the latter term conveys an image of searching existing software for code that almost does the job, then tearing it out, hacking it up, and hammering on it to fit in its new role. Crudely using a text editor in this way is likely to invalidate whatever properties the reused software was known to have. The real benefits require that the reused software components are literally the identical text.

Although we refer to software components, we recognise that possibilities for reuse are much broader. For example, software architecture, algorithms, data structures, file structures, and user interfaces could all be reused even if no code was. At the other end of the spectrum, databases, documentation, test procedures, and even test results can be reused if appropriate commonality exists. And as already mentioned, both users and developers can reuse their skills without needing retraining (and with reduced risk of butter-finger errors) if we ensure programs behave the same.

3. DOD-STD-2167A

No discussion of Defense System Software Development can ignore this bible, which together with its DIDs, sets out required activities, required deliverables, required documentation and required reports. Unfortunately, it really takes the position discussed earlier of looking at the development of isolated programs. Although software reuse is alluded to, nothing in the document really encourages its use, and the very choice of the Computer Software Unit (CSU), i.e. a separately testable element, as the basic entity in the Software Development Plan may conflict with reusable software that is structured at a different level. The documents required provide no place where reusability can be exposed.

The whole concept of Transitioning to Software Support denies the possibility of coping with evolution as discussed here.

4. The Technical Problem

What the whole activity in reuse is attempting to do, and what we want to do in evolving software, is to defy the third law of thermodynamics, or in software terms, to defy the law that structure in software degrades with time. Instead, through explicit effort, we want to improve the structure of software as we understand it better.

There are several viable approaches in use. We will discuss toolkits, program generators, problem-specific languages, configuration management systems, and object oriented programming. A key aspect of all of them is the extensibility offered as experience reveals new needs.

The oldest idea is that of the toolkit, the library of building blocks from which, possibly with additional glue, programs can be built. Originally subroutine libraries were the usual choices, but today many other resources from data structures to dialogues are collected in such libraries. The tools for finding appropriate building blocks for a particular purpose vary widely, with the browsers for class libraries of object oriented systems being the best available. But the major problems are that there is usually little automation to help design what building blocks should be put in the library, and even less automation to help combine building blocks to achieve programming objectives.

This leads to the second idea, that of the program generator. Here the library of building blocks is not used directly by the programmer, but rather by a program, the program generator, that understands how to combine them. When the objective is defined to the program generator, it will choose a combination of building blocks to achieve that objective. For instance the program generator will call buiding blocks if they are subroutines or construct them if they are database schema. While this approach can be extremely effective, the domain of problems addressed is usually quite restricted.

Between these extremes is the idea of a problem-specific language. Sometimes this language is truly independent, but more often it is a dialect of some common language, where restriction through convention to particular procedures and data structures gives the effect of a specialised language while leaving the common language as an escape to provide extensibility.

Configuration management systems are a quite different idea, and here configuration management is used in quite a different sense from that, for instance, in DOD-STD-2167A. Both uses of the term are based on the decomposition of a system into building blocks, with intervening levels of subsystems. However, whereas DOD-STD-2167A is concerned with building only one system, concentrating on the construction and integration of the building blocks to achieve that, configuration management as used here is concerned with building any system from a catalogue of related systems. Consequently, it includes the narrower sense, but goes beyond in that each particular system that can be configured is based on selecting from a choice of building blocks and subsystems, and that choice must be realised, validated and tracked.

All these ideas can be implemented with most programming languages and with many programming styles, but the current fashion of object oriented programming is particularly effective both because inheritance reduces the volume of code and because encapsulating actions

with data narrows the building block interface that must be understood and matched. A particularly important issue, however, is that of persistent objects, for often it is the values, and not just function and structure, that need to be shared.

Programs implemented on such machinery can exploit the commonality between members of a family. The extensibility of such machinery means that as requirements evolve, if the new requirements cannot be met by combining the building blocks in different ways, then the building blocks themselves can evolve to accommodate the new needs. Of course constructing the program is not the whole story. Various scaffolding, support software particular to the product being built, must also evolve. Test harnesses, regression testing suites, comparators designed to highlight differences between versions, table generators, dump analysers, performance monitors, and other specialised tools are typical examples.

Evolving the program itself, even with its supporting software, is only part of the problem. When the program is used at many installations, the transition from one release to another needs planning and possibly tool assistance. The problems encountered in field upgrade are entirely different from those encountered in shipping only new products. Cutover to an upgraded version may pose special problems, especially if it must be done without service outage. Persistent data, such as file structures or databases, may need reformatting. Phasing problems, where some sites are running on the latest release while others are running on earlier ones, increase the cost of maintenance because multiple versions must be maintained, and the situation gets much worse when individual sites can run parts of several different releases. Education and training to bring both users and maintainers up to speed on the new versions must be produced.

The foregoing discussion has referred to the result of software development as "the program". In reality, large software systems consist of suites of programs, often for different platforms used synergistically. While this complicates the technical problem, it does not fundamentally change the issues or approaches.

5. The Contractual Problem

The world of commercial shrink-wrap software shows that evolution can be accommodated economically and effectively when the specifying and implementing agency are the same. Software developed for governments, such as military or public service systems, face problems induced by the procurement process that dwarf the technical ones. Open tendering to published specifications, with the preference to lowest bidder, discourages planning for the future. Cost plus funding discourages software, or even tool, reuse. Even follow-on contracts do not ensure continuity of approach, while bringing new staff (much less new contractors) up to speed on the structure and style of existing software is hard and expensive. We do not have the mechanisms to pay contractors not to write code but to reuse it instead.

The solution to these contracting problems is not clear, but it is evident that part of the solution must be that functional specification of the currently needed system is not enough. Specification of a plan for change and a plan for reuse must be part of the contract, with the consequent specification of support and tools. The cost of identifying, evolving and certifying reusable components must be distinguished from the cost of using them, as it is for hardware.

Finding a resolution to these contractual problems is the real challenge, and is the key to coping with change.

REPORT DOCUMENTATION PAGE

1. Recipient's Reference:	2. Further Reference:
3. Originator's Reference: AC/243(Panel 11)TP/1	4. Security Classification: UNCLASSIFIED/UNLIMITED 5. Date: 15.04.91 — 6. Total Pages: 12

7. Title (NU):

AMEP: A Case Study in Object Oriented Systems Engineering

8. Presented at:

AC/243(Panel 11) Symposium on Military Information Systems Engineering - RSRE, Malvern, UK - 8-10 May 1990

9. Author's/Editor's:
Dr. Brian M. Barry

10. Author(s)/Editor(s) Address:
DREO
3701 Carling Avenue
Ottawa, Ontario K1A 0H4
Canada

11. NATO Staff Point of Contact:
Defence Research Section
NATO Headquarters
B-1110 Brussels
Belgium
(Not a Distribution Centre)

13. Keywords/Descriptors:

ESM, OBJECT ORIENTED PROGRAMMING, INCREMENTAL DEVELOPMENT, SOFTWARE ENGINEERING, SMALL TALK, PROTOTYPING

14. Abstract:

An Electronic Support Measures (ESM) signal processor may be characterised as vertically integrated: it must control low-level devices, perform computationally intensive signal processing, and may even have knowledge-based subsystems. The most difficult ESM subsystems to specify are those dominated by data-driven heuristic algorithms which are applied non-deterministically. A prototyping approach allows requirements for embedded systems of this type to be evolved in incremental stages. When compared with more traditional systems engineering methods, incremental development is not only more productive, it also results in systems which are better matched to end-user requirements. Object oriented environments provide the programming facilities needed to support prototyping and incremental development. The Defence Research Establishment Ottawa's Object Oriented Development and Test system (OODTS) is a good example of such an environment. AMEP, a prototype ESM signal processor which is being developed using the OODTS, illustrates both the benefits of incremental development and the utility of the OODTS.

AMEP: A CASE STUDY IN OBJECT ORIENTED SYSTEMS ENGINEERING

BRIAN M. BARRY*

* Defence Research Establishment Ottawa, Ottawa, Canada

1. INTRODUCTION

The Advanced Modular ESM Processor (AMEP) is a prototype ESM signal processor for naval applications, which has been developed using an object oriented methodology [1.1][1.2]. Like most large C^3I applications, AMEP is a vertically integrated system, handling everything from hard real-time data acquisition tasks to knowledge-based signal processing and complex user interfaces. An Object Oriented Development and Test System (OODTS) has been developed at DREO to provide a testbed for ESM signal processing research (an Electronic Support Measures or ESM system is a passive surveillance receiver which intercepts and analyzes radar signals). Each testbed is built using VME bus components: a programmable pre-processor which selectively captures and buffers input signal data, 4-6 single board computers which perform signal processing tasks, and peripherals to support I/O (Figure 1.1).

The OODTS integrates Smalltalk and C language tools with Harmony [1.3], a real-time multitasking multiprocessing kernel. Unlike most other multiprocessor real-time operating systems, which have been "scaled up" from uniprocessor environments, Harmony was designed specifically for multiprocessor operation. The Smalltalk dialect used in the testbed, called Actra, provides actor objects which can execute concurrently and are functionally equivalent to Harmony tasks [1.4]. Conceptually, each actor encapsulates a number of cooperating non-actor objects which execute sequentially. Actors synchronize their activities and communicate by sending one another messages. They are organized hierarchically, and use delegation to share responsibility for performing tasks. Actra has been integrated with the ENVY[1] /Manager source and object code management system, which traces its lineage to the Orwell system [1.5].

Throughout the AMEP project, we have followed an incremental approach to software development which is complemented by the OODTS. The testbed environment provides an exploratory programming system which permits developers to evolve requirements in a disciplined way as a project progresses. We begin with a simulation or emulation of the desired system, and then, supplying details as required, "grow" it by stages into a solution. In this way, design mistakes will be encountered and corrected before a large capital investment in software has been made. Since intuition is usually a poor guide for identifying performance-critical regions, we deliberately postpone optimization until the final stages of development. System performance can then be measured reliably; no guesswork is needed to identify critical regions, and no time is lost optimizing code which does not survive the full development process.

In the next section we will explain in more detail why we advocate an evolutionary approach for developing embedded systems such as AMEP. Requirements for such systems are difficult to specify, in part because they include a large number of data-driven heuristic algorithms which must be developed experimentally. A prototyping approach allows requirements to be evolved in stages, as opposed to more traditional approaches, which assume a good understanding of the problem domain and potential solutions at an early stage of development. Prototyping is often associated with object oriented design and programming methods. In the third section we will explore this relationship: we will describe the programming environment capabilities needed to support prototyping, and we will show that an object oriented system such as the OODTS has all the required facilities. To illustrate these points, we will conclude with a description of the AMEP prototype.

---

[1] ENVY is a trademark of Object Technology International, Inc.

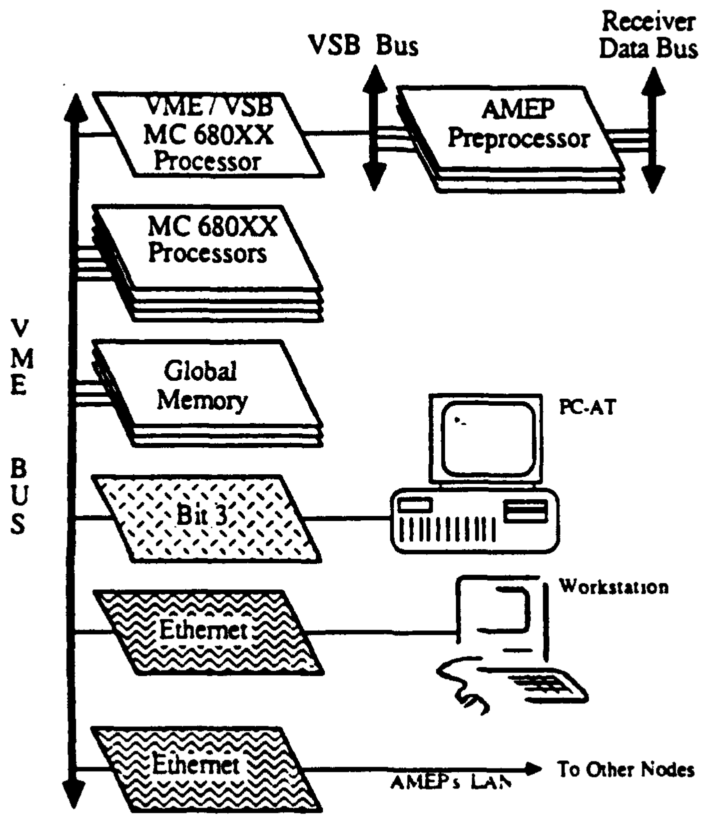


Figure 1.1: AMEP Development System

2. AN INCREMENTAL APPROACH TO REQUIREMENTS ENGINEERING

Not surprisingly, systems research for sensors like ESM tends to be driven by new demands made on the system, either by changes in the signal environment, or by the ESM system's end-users, the ESM operator and the ship's Command and Control System. Both the density (i.e. the number of radar pulses per second) and the complexity of intercepted signals can be expected to severely stress most existing ESM systems within 5-10 years. Moreover, the next generation ESM system, although it will have to be capable of autonomous operation, must also be able to function as part of an integrated sensor system. It must be able to provide the Command and Control System with high-grade information regarding the location and disposition of other platforms operating in the ship's environment.

To meet these requirements it was evident that AMEP would require much greater functionality than current systems, leading to a dramatic increase in the complexity and size of the system software. Unfortunately, repeated experience has shown that the effort required to develop large software systems does not scale up linearly [2.1][2.2]. We faced a practical problem of designing and implementing the AMEP system with very limited resources. Not only did we have to construct a working system, we also had to carry out a substantial program of research into new signal processing algorithms.

2.1 How prototyping improves productivity

It was apparent that if we could not find ways to significantly increase productivity, the scope of the undertaking would overwhelm us. Examining the so-called "software crisis" in [[2.2][2.3], Brooks asserts that the "essential difficulty" in software development lies in the intellectual exercise of conceiving software (i.e. analysis and design): getting a good understanding of the problem, finding a correct solution, and deciding how best to implement it. He characterizes software production problems (i.e. those encountered during implementation, integration, and testing) as "accidental difficulties", in the sense that there is no fundamental reason why these labour intensive tasks cannot (at least in principle) be finessed by using modern software engineering tools and techniques. This argument holds that during the analysis and design phases, the software development process is fundamentally difficult, requiring creativity and insight on the part of software engineers, and that there is really no way to avoid this.

Any strategy to improve productivity must be cognizant of these distinctions. The benefits to be gained from modern software engineering tools are felt primarily during the production stages. On the other hand, the best way to improve productivity during analysis and design is to avoid solving the wrong problem, or, at least, to catch errors early and minimize backtracking. As we shall see, incremental development combined with prototyping can help in both areas. Requirements and designs are built up in incremental steps, and then validated using an executable model, the prototype. At each stage designers are not required to understand the total solution, only the next step. Errors are detected early, making backtracking easier. The prototype itself can be built using modern programming tools such as those provided in DREO's OODTS. At each stage, the capabilities of the prototype are enhanced, until it is functionally complete. Then, by improving the real-time performance of the prototype, we can arrive at a final system solution.

2.2 Developing requirements for data-driven systems

Formulating and validating requirements is easily the most complex and demanding task encountered in new system development. It is especially difficult when there are no strong models for the system under development, e.g. when the development team has no previous experience building similar systems. This is very often the case when new sensors are being developed. Figure 2.1 shows a block diagram for a generic sensor. Sensors typically receive signal data (which they may generate themselves if they are active), and, using appropriate signal processing algorithms, measure some set of target parameters. A target tracking subsystem is usually included which groups signal data with similar parameters and provides some indication of parameter change over time. Based on this track (i.e. grouped parameter data), the system tries to infer other attributes which can be used to classify and identify the target. Nearly all sensors employ standard signal processing algorithms, which may be implemented digitally or with analogue devices, for parameter measurement and some tracking functions.

Heuristic inferencing techniques are typically used for classification and identification, and may also be used to perform tracking and pre-processing functions when there is insufficient data for classical methods (e.g. Kalman filters) to be effective. These heuristic algorithms tend to be derived empirically by studying intercepted data. In the usual case, this data is collected by the last generation sensor, and studied off-line by trained analysts who attempt to characterize it. In a traditional approach to developing new sensors, data analysis is followed by off-line algorithm development. At the end of the process, requirements and specification documents will be written which mandate the use of the most promising algorithms. Not only is this off-line development process tedious and manpower intensive, it has a serious flaw. In most instances, one needs to assess not only the performance of a

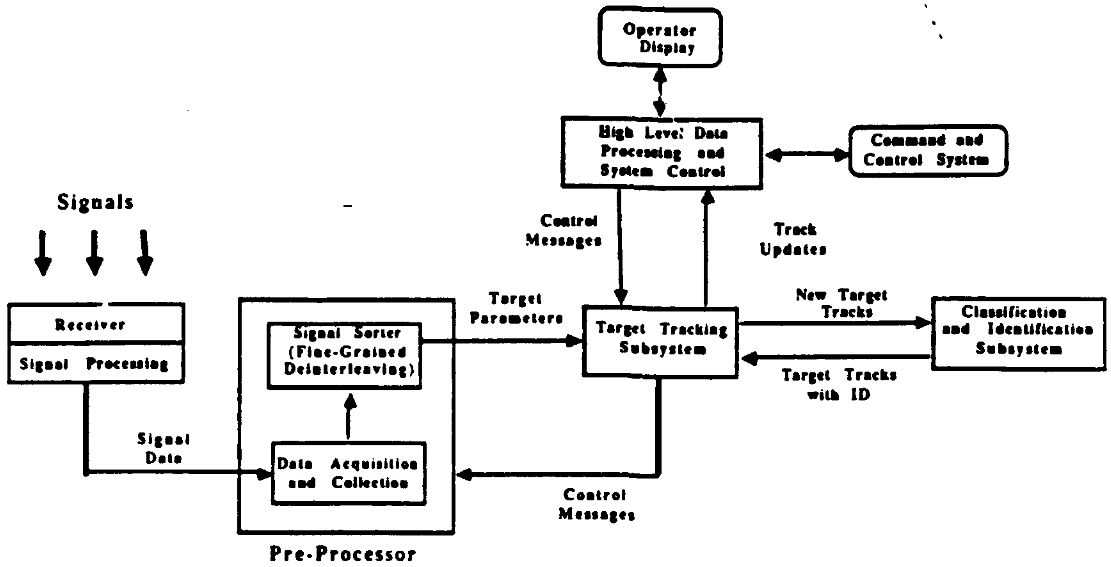


Figure 2.1: Generic Sensor Block Diagram

particular algorithm in isolation, but also the collective behaviour of algorithms, when applied non-deterministically by a data-driven decision process. For large systems, this is extremely difficult. As noted in [2.3], "the systems built today are just too complex for the mind of man to foresee all the ramifications purely by the exercise of analytic imagination...in the best modern practice, the early specification is embodied in a prototype, which the intended users can themselves drive in order to understand the consequences of their imaginings".

2.3 Prototyping complements incremental development

Prototyping techniques are ideally suited to developing ESM software requirements. Following [2.6], we define prototyping as the process of constructing software for the purpose of obtaining information about the adequacy and appropriateness of the designer's conception of a software product. Prototypes, as distinguished from production systems, are usually developed quickly, are more adaptable and expandable, and are capable of being instrumented and monitored. This last is absolutely essential: since the purpose of building the prototype is to gather information about the system under development, there must be instrumentation and monitoring tools which can collect the data needed. There is a price to be paid for this openness and flexibility, and it is usually that the prototype is (at least initially) incomplete and less efficient that a production system.

As suggested by Jacobson [2.4][2.5], there is a strong connection between a list of possible "use cases" and system requirements. In Jacobson's terminology, a use case is a particular set of events which stimulates the system, together with a set of desired responses. Use cases can be elaborated using specialization (inheritance) and composition. In principle, if all possible use cases could be determined, this would constitute a complete specification of system requirements. In the case of sensor systems, use cases will be based on combinations of possible input signals, and on operator requests for signal interpretation; in practice, the set of use cases is infinite and cannot be completely specified. In spite of this, the principal is

sound, and defining a taxonomy of representative use cases is an excellent method for analyzing requirements.

During incremental development, refining the set of use cases, and hence the system requirements, proceeds iteratively. Each prototype build may be thought of as a mini-development cycle, with its own analysis, design, implementation and test phases. The first prototype is a skeletal system which contains only a few subsystem blocks, represented in our system by actors, and fairly simple interfaces. These actors provide a framework within which signal processing algorithms can be implemented and evaluated. The initial algorithms and data structures reflect a rather naive view of the environment. As our understanding of the characteristics of the signal environment improves, our algorithms can be improved, and we are able classify and identify increasingly sophisticated signals. At the same time, by virtue of these improved signal extraction capabilities, new possibilities arise for operator interaction and data interpretation. Each stage adds new breadth or depth to the set of use cases. Testing is typically carried out using both real and simulated input data. Based on an analysis of the test results, the requirements are refined and a new mini-development cycle begins again. At each stage the prototype itself is being incrementally improved. As indicated earlier, it will eventually evolve into the final system.

2.4 Incremental development requires discipline

Based on our experience, there is a danger inherent in an incremental development approach: the process of prototype construction can build a momentum of its own, overshadowing the analysis and design activities. To counteract this tendency, it is important that there be a clear idea as to what each prototype build is meant to achieve, and how the results will be evaluated. Consequently, before the first prototype is constructed, performance goals must be established, together with criteria for determining how well these goals are met. These early performance goals are of necessity as vaguely stated as the requirements themselves, but they provide an initial place to stand. After each build, the prototype is evaluated against the performance goals, deficiencies are identified and documented (by analyzing why performance goals are not met), and a plan for changing or augmenting the prototype to remedy these deficiencies is developed. This plan guides the implementation of the next prototype. In addition, after each build the performance goals themselves as well as the evaluation criteria are also reviewed, and, if necessary, modified. When functional performance goals have been achieved, real-time performance is measured and time-critical sections of code are optimized.

3. THE OODTS AS A PROTOTYPING ENVIRONMENT

Object-oriented programming systems and prototyping frequently seem to be linked; indeed, advocates of one are very often practitioners of the other. In this section we shall see that this connection is not accidental. Based on our experiences building several AMEP prototypes, we will identify a number of the desirable capabilities which a good prototyping environment should possess. In fact, we found that the OODTS environment described in the introduction already has many of these features, and forms a good substrate on which to build the rest. We shall first consider programming language issues, and then expand the discussion to look at requirements for the programming environment as a whole.

3.1 Programming language issues

A prototyping system can be thought of as one or more programming languages together with an associated environment, that is, an integrated set of software tools. To avoid forcing particular programming language semantics on the problem during requirements analysis and design, the programming language(s) must allow creation and direct manipulation of powerful

high level abstractions. It should also provide a late binding capability. Deferring commitment is essential in the early stages of prototype development: there is no advantage to strong typing at compile time when the appropriate type has not been identified. At the same time, the prototype must be executable, so relatively low-level system programming and numerical computation must also be supported. If more than one language is used to meet these diverse requirements, they should be interoperable. To facilitate user extensions, the prototyping system should have an open architecture, in the sense that interfaces should be visible and changeable. Software reuse must be facilitated, and there should be facilities to store and retrieve persistent objects such as programs, databases, and other arbitrary data structures.

Object oriented design methods are known to result in well structured, understandable and modular systems. The prototyping system should support the object oriented programming paradigm, that is, there should be programming language support for abstract data types and inheritance. To facilitate information extraction, all data values should be first class, in the sense that they can be passed as subroutine parameters, inspected, and stored. To enable knowledge based systems to be prototyped, support should be provided for rule-based programming. There should be first class constructs for expressing concurrency. Lastly, there ought to be a large library of pre-defined classes with which to build new applications.

Actra, when augmented with C to provide low-level programming capabilities, meets most of these requirements. Since Actra is based on Smalltalk, it provides high level abstractions, dynamic binding, and an open architecture. The Smalltalk system is itself the best known example of successful software reuse, and provides a large library of well-tested reusable classes. Actra includes a Prolog compiler which is adequate for constructing simple rule bases. Actors add a modern model of concurrency to Smalltalk. Interoperability with C is provided in two ways. The first is that tools are provided to write "primitive" Smalltalk methods as C functions. Alternatively, an entire actor can be re-implemented as an independent Harmony task, which can communicate with Smalltalk actors or other tasks via message-passing. ENVY/Manager provides persistent storage of a limited set of objects associated with configuration management.

3.2 Programming environment issues

The programming environment should feature an integrated tool set, with a common representation for shared data, which includes the usual software development tools (e.g. compilers, loaders, editors, browsers, etc). In addition, it should be extendible, have accurate timing mechanisms, and employ a modern user interface. To support incremental software development, small changes should not result in long compilation times. One possibility is dynamic linking and loading of separately compiled modules; other solutions are also feasible. The environment must include garbage collection (automatic memory management); forcing programmers to worry about memory management is unacceptable overhead during prototyping. However, to allow evolution towards a real-time system, it is highly desirable that garbage collection be flexible enough to permit directives and manual overrides by the programmer. Support for multiprocessors and multitasking, as well as the standard networking protocols, is also required. To enable transparent portability, the environment should be supported on a number of standard platforms; for embedded applications, the host development system should be closely integrated with the target.

As mentioned earlier, prototypes are developed as a means of collecting information about the behaviour of the system under development. Consequently, a prototyping system must include a rich set of information extraction facilities. Such instrumentation should be as non-intrusive as possible, and should permit the end-user to collect information about static and dynamic time/space use, perform event-driven data inspection and recording, and monitor individual instances of objects. It should be possible to construct test harnesses which allow

incomplete systems to be executed, and which support test scenario generators. Data analysis and presentation tools which can be customized for the application are needed. Support must also be included for the requirements analysis and design processes. Browsers and diagramming tools should be available, as well as document and report generators. Software management tools must provide indigenous support for team programming (source and object code sharing, electronic mail). This must be done within the context of a configuration management system which provides fine-grained, multi-threaded version control.

How does the OODTS environment measure up against these requirements? To begin with, it inherits Smalltalk's extendible, integrated tool set and user interface, which is still the standard by which others are judged. Timing to microsecond accuracy has been added by accessing timer chips on the testbed's single board computers. Smalltalk already provides incremental compilation to byte codes (the underlying virtual machine is a byte code interpreter), and ENVY/Manager adds what amounts to a dynamic link capability. Garbage collection is of course provided (another Smalltalk legacy); while not yet available, support for some form of programmer directives and manual overrides is planned. Harmony provides the multiprocessor and multitasking support; TCP/IP over ethernet is also included. In the DREO testbed, code development is done on personal computers which have a wide bandwidth bus-level interface to the VME target, and are able to generate and receive interrupts.

Partial progress has been made towards satisfying the remaining requirements. To instrument target applications, the main tools are a statistical Profiler which takes snapshots of the virtual machine stack, a Performance Monitor which instruments individual methods, and encapsulators, which place a monitoring "wrapper" around individual objects. Test harnesses and data analysis tools tailored to the ESM application are planned. ENVY/Manager provides support for team programming, as well as first class documentation objects. The latter are used as input for automatic report generators which have been developed at DREO. At the time of writing, the environment is lacking design and requirements analysis tools, database-level code metric collectors, diagramming aids, and code generators.

4. THE AMEP PROTOTYPE

The AMEP project team (staffed at the 10 – 12 person level) has used the Actra environment to implement a series of AMEP prototypes over the last three years. As might be anticipated, the AMEP system design follows a strongly object-oriented approach. Object-oriented design emphasizes the importance of identifying the main objects in the problem domain and their natural interfaces, and modeling these directly in the system software. The emphasis at this stage of the research project has been on defining and enhancing the "core" objects in the system.

4.1 An overview of AMEP

The main objects in the AMEP "core" system, as shown in Figure 4.1, are:

(a) The Pre-Processor, which performs the basic deinterleaving or data reduction task. Its function is to control the AMEP pre-processor hardware, acquire new signals which enter the environment, and track data from previously identified signals.

(b) The Track, which is responsible for classification and identification of data from a single emitter in the environment. Tracks function like a mini-ESM system: they take new signal data collected by the Pre-Processor, analyze it, and attempt to match the observations with known emitters.

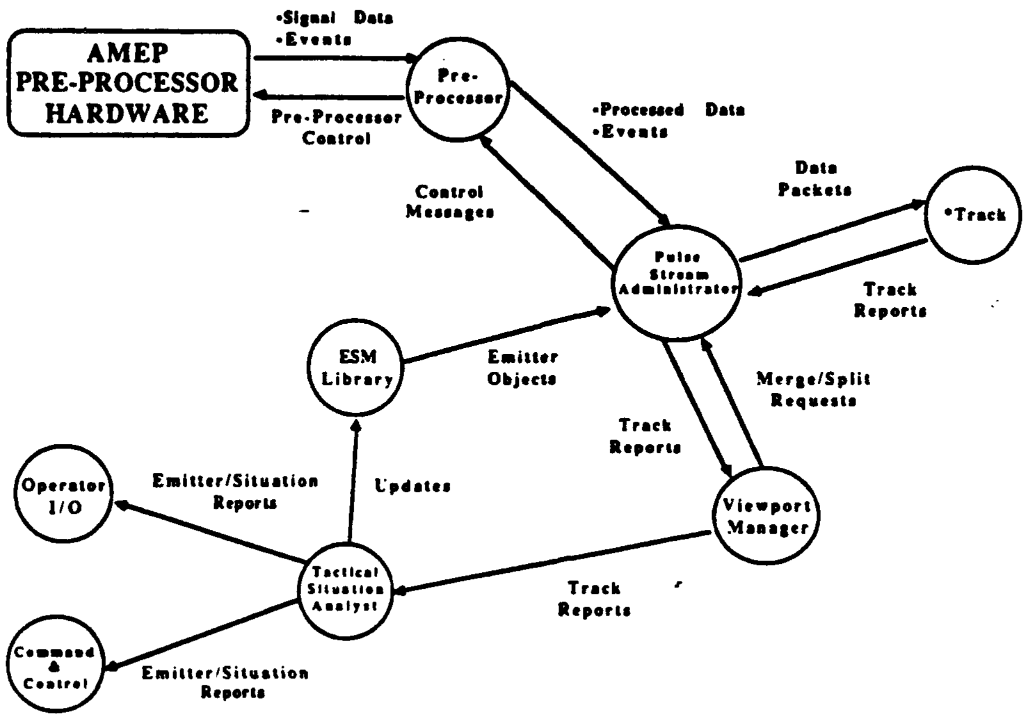


Figure 4.1: AMEP Dataflow Diagram

(c) The Pulse Stream Administrator, which provides an intelligent interface between the Pre-Processor and the Tracks. The Pulse Stream Administrator creates and manages Tracks, routes data from the Pre-Processor to the appropriate Track, and provide the Pre-Processor with information which can be used to fine-tune the pre-processor hardware.

(d) The Viewport Manager, which is responsible for coordinating all activities within a defined segment of emitter parameter space. A component, the Local Track Analyst, holds the results of Track processing on active emitter tracks. There is also software for detecting and merging fragmented tracks (i.e., tracks generated by the same emitter which should have been correlated but were not), splitting tracks, and archiving Track objects which are inactive.

(e) The ESM Library, which has the combined functionality of an object-oriented emitter database and a database management system. The Track passes high-level descriptions of new signals to the Library, which responds with candidate emitter types to explain the observations. The design of the ESM Library makes it possible for each emitter type to have its own customized data analysis software.

(f) The Tactical Situation Analyst, which tries to deduce the current tactical situation around the ship. The design permits this module to evolve into a knowledge-based expert assistant for the ESM operator.

4.2 Review of lessons learned

Table 4.1 shows code metrics for the version of the AMEP prototype reported on in [1.1] (third column), as well as its successor (second column), which was completed in early 1990.

These are contrasted with statistics for the Smalltalk/V286[2] environment. As can be seen, AMEP is approximately three times the size of the Smalltalk environment itself. The data we have collected confirms some of the"software myths" which have wide circulation in the Smalltalk community regarding the size of methods (6 - 7 lines of code) and classes (about 20 methods). We also collected data on several static measures of inheritance. We were motivated by the fact that a future version of AMEP will likely be coded in Ada, which does not support inheritance. The "average inherited classes" refers to the average depth of the inheritance tree, excluding the root class (called Object in Smalltalk). The "average inherited methods" is the total number of methods inherited from superclasses, again excluding those inherited from class Object. The rationale for excluding Object is that it provides capabilities analogous to those implemented by "system calls" in more traditional environments; we wished to measure the significance of inheritance in the application code. Our conclusion was that in certain subsystems, especially those involving complex data structures such as knowledge bases, porting to Ada may indeed cause difficulties.

During the early stages of the AMEP project, effort was focussed on identifying major ESM functions, identifying the main actors, partitioning functions amongst the actors, and understanding the nature of the resulting interactions. The early prototype designs reflected this activity: they were incomplete, and very unstable, undergoing major structural changes from one version to the next. By contrast, the effort is now directed towards improving the functionality of the system: deriving better algorithms, measuring response times, etc. Architectural changes are still occasionally necessary, but they occur much less frequently. Once again, this is reflected in the prototype. Even though the current version is nearly twice as large as its predecessor, it is structurally very similar.

	V/286 Image	Current AMEP	Last AMEP
Lines of Code	16405	46305	26341
Lines of Comment	5253	65913	37646
Lines of Documented Code	21658	112218	63987
Total Classes	115	347	218
Lines of Code/Method	7.01	6.58	5.90
Methods/Class	20.34	19.35	20.13
Average Inherited Classes	1.28	1.98	1.75
Average Inherited Methods	33.50	46.60	48.98

Table 4.1: Comparison of Code Metrics for AMEP Prototypes

Using the OODTS environment, we have measured our productivity for new code at about 4800 lines of code per man-year. This is somewhat better than what is typically reported for procedural languages like C or Ada in industrial environments, but much less than is often claimed for object oriented languages like Smalltalk. We suspect that higher numbers derive from small two or three person projects, which often do not produce industrial strength documentation, and feel that our number is more representative of what can be achieved with a large team. As a caveat, it should also be stated that it is difficult in an OOPS environment to provide hard numbers for productivity without an instrumented source code database. In

[2] Smalltalk/V286 is a registered trademark of Digitalk Inc.

particular, the impact of code reuse is hard to quantify: who is the most productive programmer, one who writes 5000 lines per year, or one who reuses 20,000 lines?

Another factor which must be accounted for is that Smalltalk achieves more functionality per line than a typical procedural language. We believe that on average, in a large system such as AMEP, one Smalltalk statement roughly translates to 4–6 lines of C or Ada , largely because of increased opportunity for reuse. Again, one must be careful interpreting such numbers. There will be a large variations between code intended for different kinds of applications; however, these variations should even out if the project is large enough and has enough diversity. Taken together, we believe that slightly increased productivity for new code production plus better reusability provides a overall productivity multiplier which is somewhere between 5 and 8. Further studies are planned to try to make this figure more precise.

5. CONCLUSION

Based on our experiences with the AMEP project, we are solidly committed to the kind of incremental system development approach described in this paper. Of course, the arguments which have been presented, and the conclusions drawn, have direct relevance only to applications which are similar to AMEP: large projects, which begin with inexact requirements, and are oriented towards embedded systems. We are also convinced that an object oriented programming system, such as the Actra environment, is the right choice for constructing such prototypes. As noted, to have built an equivalent system using a traditional approach with languages such as C or Ada would have required at least five times the effort, and the resulting system would have been much less flexible. In the final analysis, it is the economic arguments which are most persuasive: since mistakes are inevitable, one must find ways to make the cost of repairing mistakes demonstratively less than the cost of learning to live with them. Otherwise, poor systems are inevitable.

6. REFERENCES

1.1 Barry, Brian M., "Prototyping a Real-Time Embedded System in Smalltalk", Proceedings of OOPSLA 89, New Orleans, La., ACM SIGPLAN, October 1989, Pp. 255.

1.2 Barry, Brian M., D.A. Thomas, John R. Altoft, and Mike Wilson, "Using Objects to Design and Build Radar ESM Systems", Proceedings of OOPSLA 87, Orlando, Fl., ACM SIGPLAN, October 1987, Pp.192.

1.3 Gentleman, W. Morven, "Using the Harmony Operating System", National Research Council of Canada Report No. 24685, National Research Council of Canada , Ottawa, Canada, May 1985.

1.4 Thomas, David. A., Wilf R. Lalonde, John Duimovich, Michael Wilson, Jeff McAffer, and Brian Barry, "Actra - A Multitasking/Multiprocessing Smalltalk", Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, ACM SIGPLAN Notices, Volume 24, Number 4, Pp. 87.

1.5 Thomas, D.A., and Kent Johnson, "Orwell - A Configuration Management System for Team Programming", Proceedings of OOPSLA 88, San Diego, ACM SIGPLAN, September, 1988, Pp. 135.

2.1 Boehm, Barry W.,"Improving Software Productivity", IEEE Computer, Vol. 20, No.9, Pp. 43.

2.2 Brooks, Frederick P., "No Silver Bullet: Essence and Accidents of Software Engineering", IEEE Computer, Vol. 20, No.4, Pp. 10.

2.3 Brooks, Frederick P. et. al., Defence Science Board Task Force Report on Military Software, Office of the Under Secretary of Defense for Acquisition, Washington, D.C., September 1987.

2.4 Jacobson, Ivar, "Language Support for Changeable Large Real Time Systems", Proceedings of OOPSLA 86, Portland, Ore., ACM SIGPLAN, September 1986, Pp. 377.

2.5 Jacobson, Ivar, "Object Oriented Development in an Industrial Environment", Proceedings of OOPSLA 87, Orlando, Fl., ACM SIGPLAN, October 1987, Pp. 183.

2.6 Draft Report on Requirements for a Common Prototyping System, November 1988; Robert Balzer, Chairman, Richard Gabriel, Editor; Published in SIGPLAN Notices, March 1989, Pp. 93.

ACKNOWLEDGEMENTS

The author wishes to acknowledge the contributions of the AMEP project team at DREO, of the Actra developers, and of the developers of Harmony. Without their efforts, this paper quite obviously could not have been written. Several thoughtful discussions with Ivar Jacobson, Brad Cox, and Dave Thomas were very helpful in preparing the material in sections 2 and 3, although any conclusions or opinions stated are the author's alone.

NATO UNCLASSIFIED

REPORT DOCUMENTATION PAGE

1. Recipient's Reference:	2. Further Reference:
3. Originator's Reference: AC/243(Panel 11)TP/1	4. Security Classification: UNCLASSIFIED/UNLIMITED
	5. Date: 15.04.91 — 6. Total Pages: 17

7. Title (NU):

Transformational Implementation of JSD Specifications in Smalltalk-80

8. Presented at:

AC/243(Panel 11) Symposium on Military Information Systems Engineering - RSRE, Malvern, UK - 8-10 May 1990

9. Author's/Editor's:
Colin Lewis - Bryan Ratcliff

10. Author(s)/Editor(s) Address:

CA1 Division
RARDE
Fort Halstead
Sevenoaks
Kent TN14 7BP
United Kingdom

Dept. of Computer Science
Aston Triangle
Birmingham
B4 7ET
United Kingdom

11. NATO Staff Point of Contact:
Defence Research Section
NATO Headquarters
B-1110 Brussels
Belgium
(Not a Distribution Centre)

12. Distribution Statement:

Approved for public release. Distribution of this document is unlimited, and is not controlled by NATO policies or security regulations.

13. Keywords/Descriptors:

OBJECT ORIENTED PROGRAMMING, JACKSON SYSTEM DEVELOPMENT, SMALLTALK-80, TRANSFORMATIONS, IMPLEMENTATION FOLLOWSETS

14. Abstract:

The paper first presents an overview of Jackson System Development (JSD) and Object Oriented Programming. JSD is an operational software development method in which implementation is essentially a transformational process. Object Oriented Programming is a data abstraction and encapsulation paradigm with an architecture able to support software reuse. The approach adopted to map JSD specifications into procedural environments typified by Cobol, Ada, etc. involves techniques such as inversion and state vector separation. At present, however, no strategy exists for mapping JSD specifications into object oriented environments. The paper describes transformations capable of mapping JSD specifications into Smalltalk-80. The basic strategy is one whereby inversion is used to remove all concurrency in a specification. Two approaches implementing inversion are described.

TRANSFORMATIONAL IMPLEMENTATION OF JSD SPECIFICATIONS IN SMALLTALK-80™

Colin Lewis* and Bryan Ratcliff**

Contents

*CA1 Division, RARDE, Fort Halstead, Sevenoaks, Kent TN14 7BP.
**Department of Computer Science and Applied Mathematics, Aston University, Aston Triangle, Birmingham B4 7ET.

™ Smalltalk-80 is a registered trade mark of ParcPlace Systems, USA

ABSTRACT

The paper first presents an overview of Jackson System Development (JSD) and Object Oriented Programming. JSD is an operational software development method in which implementation is essentially a transformational process. Object Oriented Programming is a data abstraction and encapsulation paradigm with an architecture able to support software reuse. The approach adopted to map JSD specifications into procedural environments typified by Cobol, Ada, etc. involves techniques such as *inversion* and *state vector separation*. At present, however, no strategy exists for mapping JSD specifications into object oriented environments. The paper describes transformations capable of mapping JSD specifications into Smalltalk-80. The basic strategy is one whereby inversion is used to remove all concurrency in a specification. Two approaches implementing inversion are described. The first approach realises inversion by manipulating Smalltalk-80 *contexts* (stack frames). This is possible because contexts are first class objects which are accessible to the user like any other system object. However, problems associated with this approach are expounded. The second approach realises the behaviour of a suspend and resume mechanism via structures called 'followsets'. A followset represents all possible state transitions a process can next undergo from the state it is currently in. Followsets can be automatically generated from JSD process structures and provide a basis for transforming JSD specifications into any object oriented language.

Keywords

Object Oriented Programming, Jackson System Development, SmallTalk-80, Transformations, Implementation, Followsets.

1. INTRODUCTION

Transformations for realising Jackson System Development (JSD) specifications in procedural languages such as Pascal, Fortran, Cobol, etc. are well known [1]. The aim of this paper is to describe a transformational approach for implementing JSD specifications in object oriented programming languages such as Smalltalk-80 [2]. The paper:

- introduces the general characteristics of JSD and Object Oriented Programming;
- shows how JSD specifications can be transformed into Smalltalk-80 implementations.

The transformations described are not specific to Smalltalk-80, but can be used for implementing specifications in any object oriented language. Further, the transformations are also suitable for implementation in languages which are 'goto-less' (eg Occam™). The effect is to extend significantly the scope of JSD in terms of the implementation architectures to which it can be targeted.

The paper is divided into five sections. Section 2 introduces the JSD method. Section 3 briefly describes Object Oriented Programming and Smalltalk-80. Section 4 describes two transformational approaches for implementing specifications in Smalltalk-80. Finally, Section 5 offers some brief conclusions.

2. JACKSON SYSTEM DEVELOPMENT

2.1 Background

JSD is a method for developing software systems. The method covers much of the software life-cycle, starting from an existing system requirement and proceeding through to a fully implemented system (Figure 1).

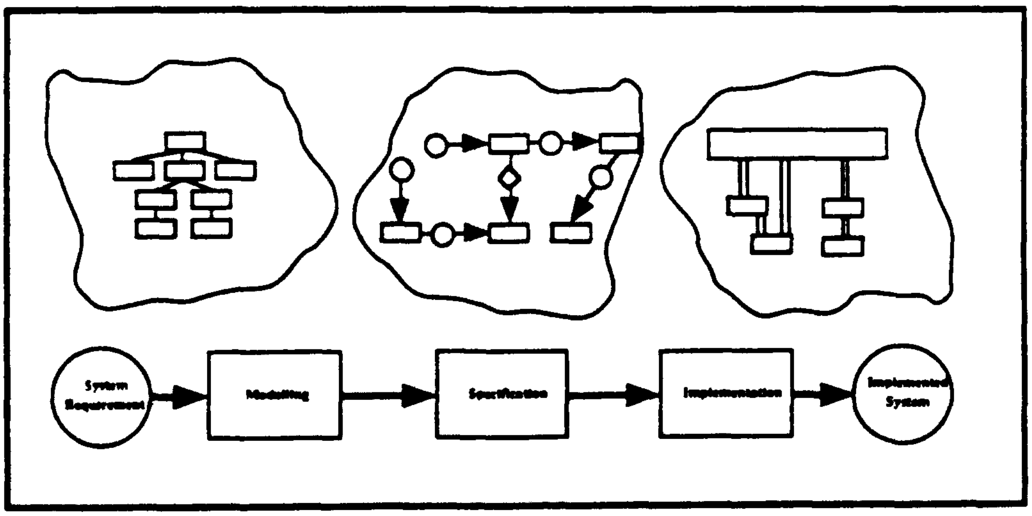


Figure 1. JSD Overview

The approach is systematic in that it decomposes the development task into well defined stages. In particular, there is a distinct separation between specification and implementation [3]. The starting point in JSD is the development of a model of the real

™ OCCAM is a registered trade mark of the INMOS group of companies

world subject matter on which an abstract system specification will be based. Functions are then built upon that model to produce the required input/output activities of the system. Finally, the specification is transformed to produce an implementation of acceptable performance.

2.2 Modelling

The model is a representation of the real world within which the system to be built exists [4,5]. "The JSD insistence on starting development by explicitly modelling the real world ensures that the system user's view of reality is properly embodied in the specification and, eventually, in the structure of the system itself." [1] The real world model scopes a system by implicitly defining what possible functions the system can support. The model itself is represented by a set of disconnected sequential processes. A sequential process expresses the possible time ordering of external real world events, called *actions*, using three basic components: sequence, iteration and selection. A time-ordered set of actions which some real world object suffers or performs is called an *entity life history* and represents all possible (valid) orderings of the real world actions. Actions generate messages which are read by the system's model processes, which themselves correspond to the entity life history models. These processes then (partially) execute and thus synchronise themselves with the dynamics of the outside real world (albeit always inevitably lagging behind [1]). In addition, all data about which the system is to provide information is local to these model processes.

2.3 Specification

The system model on its own provides only an abstract simulation of the real world subject matter under investigation [4]. Functional processes are connected to model processes producing either system outputs or events not explicitly available in the real world; the latter generate additional inputs to the model processes. Other processes capture data generated by the real world actions and then, after error checking, pass it on to the relevant model processes. Processes are connected to, and communicate with, each other by reading from and writing to idealised FIFO buffers called *datastreams*, and by inspecting each other's internal states or *state vectors*. Datastream communication is asynchronous; a process is not blocked on writing a record to a datastream, but a process is blocked when reading from a datastream that is empty. State vector inspection, which provides an additional means of communication, is a read-only access mechanism that permits one process to inspect the state of another. The inspection itself does not interrupt the inspected process.

Overall, the specification phase produces a network of asynchronously communicating sequential processes. This network is essentially an abstract system architecture, the other main constituents of the specification being the descriptions of processes. An important characteristic of a JSD specification is that it is in principle executable using suitable interpreters. This *operational* characteristic of JSD specifications [6] can give developers an early preview of how the system will behave when implemented. Operational approaches such as JSD can therefore lead potentially to a reduction in the overall cost of development since verification of a system's functional behaviour can take place at an earlier stage than usual [7].

2.4 Implementation

JSD uses a system specification directly to generate the desired implementation [4]. This transformational approach ensures that the integrity of the specified system does not become corrupted during the implementation phase [1]. Two major transformation techniques used in JSD are *inversion* and *state vector separation*. There are other transformational techniques, such as *dismemberment*, but discussion of these lies outside the scope of this paper.

Inversion, which is typically applied in conjunction with state vector separation, in its basic form converts an asynchronous producer-consumer process pair into a routine and re-entrant subroutine that are behaviourally equivalent to two coroutines [8,9]. When the re-entrant subroutine is invoked, it resumes execution from where it last left off. Partial execution takes place, updating internal states, until the process suspends itself. The re-entrant subroutine's suspend points are associated with the original read (or write) operations on its connected datastream, which the inversion transformation removes.

JSD specifications often contain many instances of a process type, all of which are executing concurrently. They are each identical in structure, but their local states will be different. These many instances can be implemented by having one copy of the process text and many copies of the state vectors. Thus, each process is implemented by separating its state vector from its sequential procedure.

State vector separation and inversion enable an entire specification to be implemented on a single machine if desired. Each instance of a process type can run on the same processor by storing its many state vectors in a 'database' and using the stateless re-entrant subroutine to update a loaded state vector during its execution; at the next suspend point the (possibly updated) state vector is written back to the database. To make sure that each process in the specification gets its proper share of processor

time, special-purpose scheduler processes are designed, which implement any special timing constraints that the system must satisfy.

3. OBJECT ORIENTED PROGRAMMING

This brief overview of object oriented programming is based on Smalltalk-80, since Smalltalk-80 was the first system to popularise the object oriented paradigm [10] and is the implementation environment for the transformations described in Section 4. The following features can be said to typify an object oriented system:

i. Objects
ii. Inheritance
iii. Message Passing
iv. Persistence

3.1 Objects

Encapsulated data abstractions minimise interdependencies among separately written modules by defining strict external interfaces [11]. The more rigorous the encapsulation mechanism used in a system, the less chance that changing one part of it will adversely affect another. The encapsulation mechanism in object oriented systems is the *object*. An object is composed of two parts: an object state, represented by some internal variables called *instance variables*; and functionality, represented by a set of procedures or *methods*. A method is a modularised set of operations which normally operates on the instance variables of the object in which the method resides. From the Smalltalk-80 point of view, data abstractions and encapsulation are knitted together very closely. The encapsulation mechanism enforces interaction with an object via its external interface or *protocol*.

3.2 Inheritance

In Smalltalk-80 and most other object oriented languages, a *class*, which is the templating mechanism for creating objects, is always a subclass (and specialisation) of another class. For this subclassing mechanism to function, the classes are (usually) arranged in a hierarchy in which every object is an instance of just one class. This is *single inheritance*. *Multiple inheritance* is a generalisation that allows a given class to have more than one immediate superclass (i.e. the inheritance hierarchy is not a pure tree). In Smalltalk-80, all objects inherit properties from class `Object` at the top of the hierarchy, and hence `Object` describes the default behaviour of all objects [12].

Inheritance enables a programmer to adopt a software reuse approach in writing programs, creating new objects from existing ones. "The fundamental idea of inheritance is that new software elements may be defined as extensions of previously defined ones; existing elements do not have to be modified when used as a basis for new definitions." [13]

3.3 Message Passing

Message passing is pre-requisite in overcoming the inevitable combinatorial explosion of routine complexity in extendable polymorphic systems. Conventionally, a routine is polymorphic when it is type-generalised. This ability to abstract over types [14] enables programmers to produce generalised software components [15]. With the development of the message passing metaphor, the burden of doing explicit type checking and type dispatching disappears by making the routines themselves monomorphic and embedding them within system types (i.e. classes) [16]. Within this metaphor, all processing activity is initiated simply by sending messages to objects; processing activity proceeds inside objects themselves [19]. In Smalltalk-80, this object-message metaphor is used uniformly throughout the system, giving a simple but very powerful approach to programming [15,18].

3.4 Persistence

The persistent model of data underlies many current object oriented programming languages [17]. *Persistence* is an abstraction over the time that a piece of data is required and usable [14]. In conventional programming systems, the persistence of each data item has to be specifically handled by the programmer via the two persistent mediums available, files and databases. The separation of long-term from short-term data (stored in program variables) incurs a cost overhead in the development of large scale systems [19]. This overhead is reflected by the fact that files are unable successfully to store structured entities which reside in dynamic memory [20]. Usually a transformation is required to 'flatten' dynamic memory structures in order to store them in files.

Smalltalk-80 has a weak form of persistence in the form of its 'virtual image' in which all objects persist. This enables object states to be preserved between sessions of execution. However, in order to guarantee such persistence, the virtual image has to be explicitly saved ('snapshot'), by writing to disc a complete binary copy of the dynamic memory [21,22].

4. TRANSFORMATIONAL STRATEGY AND ITS IMPLEMENTATION

4.1 Overview

The main decision with which JSD specifications present the implementer is whether to remove the concurrency expressed therein [23], depending upon the implementation environment. Smalltalk-80 provides an object of class `Process` and so a concurrent implementation would appear to be the logical choice. However, Smalltalk-80 was not specifically designed to build highly parallel systems (hence the development of ConcurrentSmalltalk [24]) such as those described by JSD specifications. A second reason for not adopting a concurrent implementation approach using the Smalltalk-80 `Process` class is that a transformational strategy should, if possible, be sufficiently general to achieve implementation in any object oriented language. Since most languages do not possess process objects, an alternative to generating concurrent implementations is desirable. These two reasons make concurrency removal from specifications necessary.

As regards state vector separation, the behaviour of object oriented systems reflects this transformation directly. Each time a new object is created (an instance of a class), its behaviour (set of methods) is not duplicated because the methods reside in the object's class. Object creation simply makes a copy of its class's instance variables. This instantiation mechanism parallels the desired effect of state vector separation in procedural implementations.

As stated earlier, program inversion architecturally changes a network of communicating sequential processes into a hierarchy of re-entrant subroutines, i.e. coroutines. However, most languages including Smalltalk-80 do not possess a coroutine facility [25], and so the latter has to be simulated using available language constructs. As a result, conventional realisations of inversion necessitate the extensive use of the 'goto' statement [8]. Smalltalk-80, however, does not possess this construct. Nevertheless, inversion can be realised in Smalltalk-80 and the next two sections describe two possible approaches. The descriptions of these two approaches are brief and therefore necessitate some working knowledge of Smalltalk-80.

4.2 Smalltalk-80 Context Manipulation

4.2.1 Architecture and Operation

Architecturally, Smalltalk-80 has two major components: its 'Virtual Machine' (VM) and 'Virtual Image' (see 3.4). The virtual image is the static representation of Smalltalk-

80's data structures in the form of objects, while the VM brings the virtual image to life by interpreting its methods [2]. To achieve method interpretation, two representations are used: the first, a symbolic format, is the textual definition of methods specified by a user; the second is a compiled version of these definitions. Compiled methods generated by the Smalltalk-80 compiler are streams of byte codes suitable for interpretation by the VM. Every class in the system has a dictionary of all its compiled methods. The dictionary is composed of a series of key-value pairs. When a message is sent to an object, the VM identifies the class of the receiver object and then starts searching that class's method dictionary for the key specified by the message. When the key is found, the VM interprets the compiled method associated with that key.

Smalltalk-80's use of the object/message metaphor gives rise to the unusual feature of making the VM's runtime state (consisting of method activations or *continuations* [26]) visible to the programmer as data objects [27]. In Smalltalk-80 terminology, continuations are called *contexts* [2]. When a message is sent to an object a new context is created by the VM for the associated method activation. Since all processing throughout the system is accomplished by sending messages, there will be many contexts in the system at any one time. The context associated with the method currently being evaluated is called the *active* context. When the method associated with the active context evaluates a message send to an object, the active context is suspended and a new context is created and made active. The active context stores the context which activated it; the latter is called the active context's *sender* (which is akin to a procedure's caller in procedural languages). Sender contexts resume when active contexts terminate. Access to active contexts is possible via the pseudo-variable `thisContext` [2]. (NB. pseudo-variables are variables available in all methods but which cannot have anything assigned to them eg. `self, super, true, false, nil`). Contexts held in `thisContext` are *first-class* objects and so can be treated like any other object in the system. They offer a protocol allowing inspection/alteration of program counter, stack pointer, changing of sender, etc. and are used extensively by the system debugger. By manipulating contexts, it is possible to build an object which behaves like a re-entrant procedure [26]. The following describes how this is done.

4.2.2 Re-entrant Procedures

Consider the following Smalltalk-80 code. The method `controlMethod` sets a counter to zero and then creates a new instance of class `ReEntrantObject`; this instancehas the message `loop` sent to it. Once control returns back to `controlMethod`, there follows an iteration which increments the counter per repetition; the important statement in the iteration is where the instance of `ReEntrantObject` is sent the message `resume`. `Loop` is a method defined in `ReEntrantObject` which simply sets a counter to

zero and then iterates forever; the important construct here is the suspend construct. Loop returns to controlMethod at the point self suspend; it will continue execution whenever resumed from controlMethod immediately after self suspend. The suspend method returns control to its sender but saves its current context. The resume method loads a saved context and then continues execution with the loaded context.

```
controlMethod
| rObject count |
count ← 0.
rObject ← ReEntrantObject new.
rObject loop.
[count < 10]
   whileTrue:
      [rObject resume.
      count ← count + 1]
```

```
loop
| count |
count ← 0.
[true]
   whileTrue:
      [count ← count + 1.
      self suspend]
```

Implementation of suspend (in class ReEntrantObject) saves a modified copy of the current context in an instance variable of class ReEntrantObject called savedContext; the modification effected is to increment the saved context's program counter by one. Method resumption using resume is implemented as follows. Sending the message resume to an instance of ReEntrantObject creates a new resume method context. This newly created context has its sender context changed to the one saved in savedContext. Thus, when the resume method returns, it will not return to the sender context in which it was activated, but to the point after the suspend message send in the saved context.

4.2.3 Problems

In theory, the two methods suspend and resume should enable the realisation of the inversion transformation. However, when these are implemented in Smalltalk-80, sending resume to an instance of ReEntrantObject many times causes the VM consistently to crash. The exact reason for this unwanted behaviour is still unknown. One possible explanation is in the way contexts are handled in the VM implementation [28]. In order for Smalltalk-80's performance to be acceptable, the VM uses some special optimisations relating to context manipulation (the optimisation gives an eight-fold increase in VM performance [27]). Instead of creating a new context object at every message send, the VM creates a standard procedure activation which is pushed onto its internal stack [29,30]. When a method returns, the internal activation is popped off the stack. Only when a context needs to be explicitly used does the VM convert it to a real object. This optimisation of having multiple representations of contexts [27] in the VM is being exposed in the implementation of re-entrant objects as described here. The conclusion to be drawn is that although contexts are in principle first-class objects, as implemented

they are not.

4.3 Followsets

4.3.1 Followset Definition

The term *followset* is the name given to the collection of all valid state transitions a process can next undergo from any given state [31]. It is possible for the complete semantics of a process, as depicted in structure diagrams (see Figure 2), to be represented by followsets.

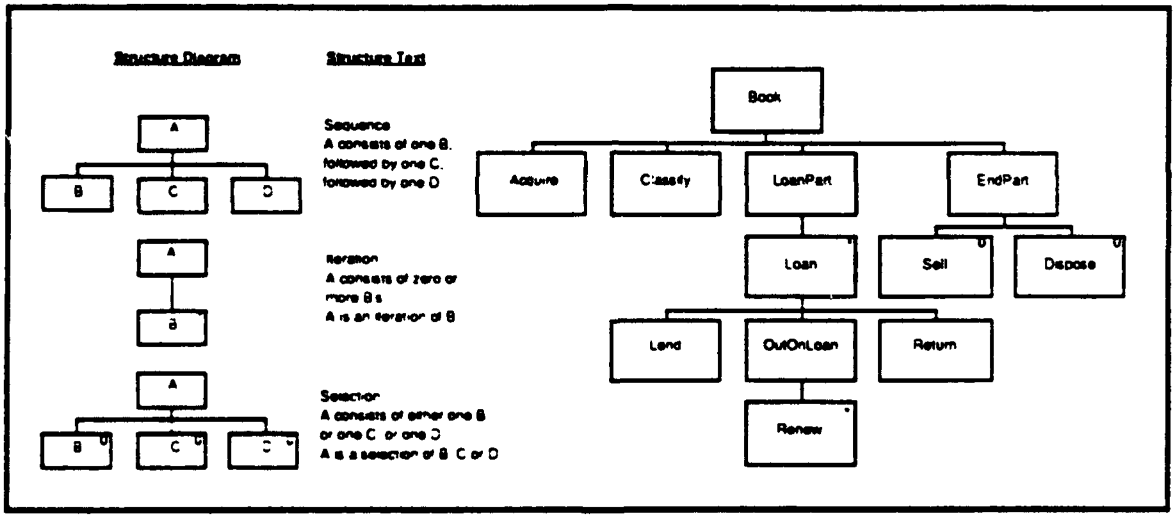


Figure 2. Structure of a process 'Book'

```
The rules -

a    If the node is a leaf node (i.e. no children) then
          FIRST(Parent) = {Parent}
b    Sequence components
     i    FIRST(Parent) = FIRST(Child1)
     ii   FOLLOW(Childlast) = FOLLOW(Parent)
     iii  FOLLOW(Childi) = FIRST(Child[illegible]) [illegible] last-1)
c    Selection components
     i    FIRST(Parent) = FIRST(Child1) or FIRST(Child2) or ...FIRST(Childlast)
     ii   FOLLOW(Childi) = FOLLOW(Parent) [illegible] last)
d    Iteration components
     i    FIRST(Parent) = FIRST(Child1) or FOLLOW(Parent)
     ii   FOLLOW(Child1) = FIRST(Child1) or FOLLOW(Parent)
e    If the node is a null leaf node then
          FIRST(Parent) = FOLLOW(Parent)
f    If the node is the root node then
          FOLLOW(Parent) = Ø
```

Figure 3. Definition of FIRST and FOLLOW

The technique for followset generation from structure diagrams is taken from the

development of predictive parsers in compiler theory [32]. Basically, two functions FIRST and FOLLOW are repeatedly applied to each node in a structure diagram, generating sets of possible leaf node states. FIRST and FOLLOW's definition are different for each node type and are in the form of calls to each other (see Figure 3). From the followset trace shown in Figure 4, which is obtained by applying the rules of Figure 3 to the process structure in Figure 2, it is possible to deduce the next valid set of states to follow any given state in the 'Book' process by evaluating FOLLOW(givenState). This ability to derive systematically the next valid state set can be used as the basis for implementing inversion.

```
FIRST(BOOK)                      = FIRST(ACQUIRE)                         - rule bi
FIRST(ACQUIRE)                   = {ACQUIRE}                              - rule a
FOLLOW(ACQUIRE)                  = FIRST(CLASSIFY)                        - rule biii
FIRST(CLASSIFY)                  = {CLASSIFY}                             - rule a
FOLLOW(CLASSIFY)                 = FIRST(LOANPART)                        - rule biii
FIRST(LOANPART)                  = FIRST(LOAN) or FOLLOW(LOANPART)        - rule di
        FIRST(LOAN)              = FIRST(LEND)                            - rule bi
        FIRST(LEND)              = {LEND}                                 - rule a
        FOLLOW(LOANPART)         = FIRST(ENDPART)                         - rule biii
        FIRST(ENDPART)           = FIRST(SELL) or FIRST(DISPOSE)          - rule ci
                FIRST(SELL)      = {SELL}                                 - rule a
                FIRST(DISPOSE)   = {DISPOSE}                              - rule a
FOLLOW(LEND)                     = FIRST(OUTONLOAN)                       - rule biii
FIRST(OUTONLOAN)                 = FIRST(RENEW) or FOLLOW(OUTONLOAN)      - rule di
        FIRST(RENEW)             = {RENEW}                                - rule a
        FOLLOW(OUTONLOAN)        = FIRST(RETURN)                          - rule biii
        FIRST(RETURN)            = {RETURN}                               - rule a
FOLLOW(RENEW)                    = FIRST(RENEW) or FOLLOW(OUTONLOAN)      - rule dii
        FIRST(RENEW)             = {RENEW}                                - rule a
        FOLLOW(OUTONLOAN)        = FIRST(RETURN)                          - rule biii
        FIRST(RETURN)            = {RETURN}                               - rule a
FOLLOW(RETURN)                   = FOLLOW(LOAN)                           - rule bii
FOLLOW(LOAN)                     = FIRST(LOAN) or FOLLOW(LOANPART)        - rule dii
        FIRST(LOAN)              = FIRST(LEND)                            - rule bi
        FIRST(LEND)              = {LEND}                                 - rule a
        FOLLOW(LOANPART)         = FIRST(ENDPART)                         - rule biii
        FIRST(ENDPART)           = FIRST(SELL) or FIRST(DISPOSE)          - rule ci
                FIRST(SELL)      = {SELL}                                 - rule a
                FIRST(DISPOSE)   = {DISPOSE}                              - rule a
FOLLOW(SELL)                     = FOLLOW(ENDPART)                        - rule cii
FOLLOW(ENDPART)                  = FOLLOW(BOOK)                           - rule bii
FOLLOW(BOOK)                     = e                                      - rule f
FOLLOW(DISPOSE)                  = FOLLOW(ENDPART)                        - rule cii
FOLLOW(ENDPART)                  = FOLLOW(BOOK)                           - rule bii
FOLLOW(BOOK)                     = e                                      - rule a

Hence the following is the trace of the structure :-

FIRST(BOOK)                      = {ACQUIRE}
FOLLOW(ACQUIRE)                  = {CLASSIFY}
FOLLOW(CLASSIFY)                 = {LEND, SELL, DISPOSE}
FOLLOW(LEND)                     = {RENEW, RETURN}
FOLLOW(RENEW)                    = {RENEW, RETURN}
FOLLOW(RETURN)                   = {LEND, SELL, DISPOSE}
FOLLOW(SELL)                     = e
FOLLOW(DISPOSE)                  = e
```

Figure 4. Followset trace of the 'Book' process

4.3.2 Guards

Associated with each state in every set of states is a *guard* representing the conditional operations associated with selections and iterations in a process's structure. Guards are necessary in order to resolve which actual state in a set of states is to follow a given state. For example, FOLLOW(CLASSIFY) = {LEND, SELL, DISPOSE}; the members of this followset have three guards g(LEND), g(SELL) and g(DISPOSE) respectively. Each guard when evaluated indicates whether or not the associated state is

the next valid one. Because JSD process semantics are completely deterministic, it is not possible for two or more guards associated with a followset to be true simultaneously. By a simple modification to FIRST and FOLLOW, guards can be derived automatically in situ of followset generation.

Followsets and their associated guards possess operational semantics similar to re-entrant subroutines. Via a simple algorithm or execution harness, the behaviour of a re-entrant routine can be simulated.

4.3.3 Smalltalk-80 realisation

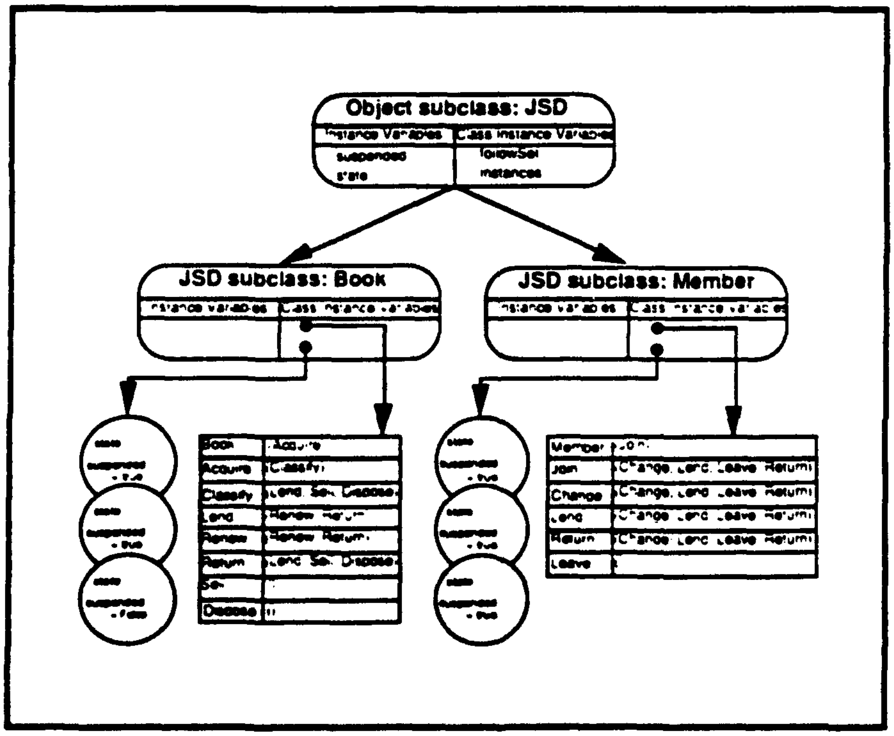


Figure 5. JSD processes realised as Smalltalk-80 classes

Process types in JSD specifications are realised as Smalltalk-80 classes [33]. These classes all have user-defined primitive operations (associated with leaf nodes in structure diagrams) and conditions (associated with selection and iteration nodes) as methods, together with a set of methods for guards. These classes have two class instance variables, `instances` and `followset`, to store the instances of the class and the inverted process's followset representation respectively - see Figure 5. Each class instance has two instance variables, `suspended` and `state`, denoting whether the process can run and

what its current state is.

Although communication is by the standard Smalltalk-80 message sending mechanism, in order to achieve process execution using the harness, a universal message interface needs to be used. Thus, all inverted process-to-process communication is accomplished by a `write` message and its derivatives. Using that message, inverted processes write to each other via their classes, since the latter are the repository for all instances. When a class receives a `write` message, the following takes place:

❶ Find the appropriate process instance (specified in the `write` message parameters)

❷ Using the process's current state (held in `state`), access the relevant followset (held in the class instance variable `followset`)

❸ For each state in the set, evaluate the associated guard functions until one returns true

❹ For the state which has a true guard, evaluate the code associated with that state

❺ Set the process's state to the new state

❻ Repeat steps ❷ to ❺ until the process suspends (instance variable `suspended` becomes true - see below).

When inversion is realised in conventional languages, the usual technique adopted is to replace read and write statements with returns and calls. Using a highly polymorphic environment such as Smalltalk-80, the technique adopted is to change the semantics of read and write. This results in read messages simply setting the process's instance variable `suspended` to true and the write messages invoking the execution harness. Thus, although read and write messages appear in code evaluated at step ❹, the act of reading and writing in the sense of asynchronous process communication has disappeared.

As regards state vector inspection, the implementation of this mechanism in object oriented systems is reduced to objects simply returning themselves via the `self` pseudo-variable; all that needs to be provided is access to an object's instance variables. A process needing access to all state vectors of another process sends a `getSV` request to the process's class; the class simply returns the entire collection of instances of itself. However, when a `getSVof:` request is sent to a class, the actual instance specified by a parameter of this request is returned.

5. CONCLUSIONS

Of the two transformational systems described in this paper, namely context manipulation and followsets, the former is the more efficient since it does not need any type of harness in which to operate (except for the virtual machine). Context manipulation as a means of achieving inversion is not a general implementation, however, since most languages do not permit manipulation of procedure activation states. In contrast, followsets are a general implementation of inversion, with the added advantage of not having to rely on the 'goto' primitive. As such, therefore, followset transformations can also be used to good effect when implementing JSD specifications in languages such as Occam.

The construction of transformations mapping JSD specifications into object oriented architectures such as Smalltalk-80 leads to two additional capabilities:

- Direct implementation of specifications in object oriented languages like Smalltalk-80;
- Rapid specification prototyping.

As regards the latter, Smalltalk-80 has been viewed as a rapid prototyping environment [34,35]. Given that it is now possible to transform JSD specifications into this environment, the behaviour of such specifications can be observed early on in their development. Such a facility realises, to considerable advantage, the truly operational nature of JSD [6,7].

6. REFERENCES

1 Jackson M.A. 'System Development'. Prentice-Hall, 1983.

2 Goldberg A. & Robson D. 'Smalltalk-80. The language and its implementation'. Addison Wesley, 1983.

3 Renold A. 'Jackson System Development for Real Time Systems', Scientia Electrica (Switzerland), vol. 34, no. 2, pp. 3-43, 1988.

4 Cameron J. 'An Overview of JSD', IEEE Transactions on Software Engineering, vol. SE-12, no. 2, pp. 222-240, 1986.

5 Cameron J. 'The Modelling Phase of JSD', Information and Software Technology (UK), vol. 30, no. 6, pp. 373-383, 1988.

6 Zave P. 'The operational approach versus the conventional approach to software development', Communications of the ACM, vol. 27, no. 2, pp. 104-118, 1984.

7 Agresti W.W. 'What are the new Paradigms', in: New Paradigms for Software Development, IEEE Computer Society, pp. 6-10, 1986.

8 Storer R. 'Data-driven software design using inversion', Information and Software Technology (UK), vol. 30, no. 2, pp. 99-107, 1988.

9 Sanden B. 'An Entity-Life Modeling Approach to the Design of Concurrent Software', Communications of the ACM, vol. 32, no. 3, pp. 330-343, 1989.

10 Rentsch T. 'Object oriented programming', ACM SIGPLAN Notices, vol. 17, no. 9, pp. 51-57, 1982.

11 Snyder A. 'Encapsulation and Inheritance in Object-oriented programming languages', Proceedings of OOPSLA'86, ACM SIGPLAN Notices, vol. 21, no. 11, pp. 38-45, 1986.

12 Ingalls D.H.H. 'Design principles behind Smalltalk', BYTE, vol. 6, no. 8, pp. 286-298, 1981.

13 Meyer B. 'Genericity versus Inheritance', Proceedings of OOPSLA'86, ACM SIGPLAN Notices, vol. 21, no. 11, pp. 391-405, 1986.

14 Morrison R., Brown A.L., Carrick R., Connor R.C.H., Dearle A. & Atkinson M.P. 'Polymorphism, persistence and software re-use in a strongly typed object-oriented environment', Software Engineering Journal, vol. 2, no. 6, pp. 199-204, 1987.

15 Harland D.M. 'Polymorphic Programming Languages - design and implementation', Ellis Horwood, 1984.

16 Ingalls D.H.H. 'A Simple Technique for Handling Multiple Polymorphism', Proceedings of OOPSLA'86, ACM SIGPLAN Notices, vol. 21, no. 11, pp. 347-349, 1986.

17 Wolczko M. 'Semantics of Object Oriented Languages', PhD Thesis, University of Manchester, 1988.

18 Ungar D. & Smith R.B. 'Self: The Power of Simplicity', Proceedings of OOPSLA'87, ACM SIGPLAN Notices, vol. 22, no. 12, pp. 227-242, 1987.

19 Atkinson M.P., Bailey P.J., Chisholm K.J., Cockshott P.W. & Morrison R. 'An Approach to Persistent Programming', The Computer Journal, vol. 26, no. 4, pp. 360-365, 1983.

20 Harland D.M. 'REKURSIV object oriented computer architecture', Ellis Horwood, 1988.

21 Low C. 'A Shared, Persistent Object Store', in: ECOOP'88, Proceedings of the Second European Conference on Object-Oriented Programming, pp. 390-410, 1988.

22 Straw A., Mellender F. & Riegel S. 'Object Management in a Persistent Smalltalk System', Software Practice and Experience, vol. 19, no. 8, pp. 719-737, 1989.

23 Hull M.E.C., Zarea-Aliabadi A. & Guthrie D.A. 'Object-oriented design, Jackson system development (JSD) specifications and concurrency', Software Engineering Journal, vol. 4, no. 2, pp. 79-86, 1989.

24 Yokote Y. & Tokoro M. 'Concurrent Programming in ConcurrentSmalltalk', in: Object Oriented Concurrent Programming, pp. 129-158, MIT Press, 1987.

25 Haynes C.T., Friedman D.P. & Wand M. 'Obtaining Coroutines with Continuations',

Computer Languages, vol. 11, no. 3/4, pp. 143-153, 1986.

26 Haynes C.T. & Friedman D.P. 'Embedding Continuations in Procedural Objects', ACM Transactions on Programming Languages and Systems, vol. 9, no. 4, pp. 582-598, 1987.

27 Deutsch L.P. & Schiffmann A.M. 'Efficient Implementation of the Smalltalk-80 System', ACM SIGACT/SIGPLAN Proceedings of the Eleventh Annual Symposium on the Principles of Programming Languages, pp. 297-302, 1984.

28 Moss J.E.B. 'Managing Stack Frames in Smalltalk', Proceedings of the SIGPLAN'87 Symposium on Interpreters and Interpretive techniques, ACM SIGPLAN Notices, vol. 22, no. 7 , pp. 229-240, 1987.

39 Miranda E. 'BrouHaHa - A Portable Smalltalk Interpreter', Proceedings of OOPSLA'87, ACM SIGPLAN Notices, vol. 22, no. 12, pp. 354-365, 1987.

30 Baden S.B. 'Low-Overhead Storage Reclamation in the Smalltalk-80 Virtual Machine', *in:* 'Smalltalk-80 Bits of History, Words of Advice', pp. 331-342, Addison Wesley, 1984.

31 Stirling C. 'Follow Set Error Recovery', Software Practice and Experience, vol. 15, no. 3, pp. 239-257, 1985.

32 Aho A.V. & Ullman J.D. 'Principles of Compiler Design'. Addison Wesley, 1979.

33 Jackson M.A. 'Information Systems: Modelling, Sequencing and Transformations', in: On the construction of programs, pp. 319-341. Cambridge, 1980.

34 Alexander J.H. 'Exploratory Application Development using Smalltalk', Tektronix Computer Research Laboratory Technical Report no. CR-85-16, 1985.

35 Diederich J. & Milton J. 'Experimental Prototyping in Smalltalk', IEEE Software, vol. 4, no. 3, pp. 50-64, 1987.

NATO UNCLASSIFIED

REPORT DOCUMENTATION PAGE

1. Recipient's Reference:	2. Further Reference:
3. Originator's Reference: AC/243(Panel 11)TP/1	4. Security Classification: UNCLASSIFIED/UNLIMITED 5. Date: 15.04.91 — 6. Total Pages: 10
7. Title (NU): Formal Specification and Requirements	
8. Presented at: AC/243(Panel 11) Symposium on Military Information Systems Engineering - RSRE, Malvern, UK - 8-10 May 1990	
9. Author's/Editor's: Patrick R.H. Place - William G. Wood	
10. Author(s)/Editor(s) Address: Software Engineering Institute (Sponsored by the US DoD) Carnegie Mellon University Pittsburgh PA 15213-3890 United States	11. NATO Staff Point of Contact: Defence Research Section NATO Headquarters B-1110 Brussels Belgium (Not a Distribution Centre)
12. Distribution Statement: Approved for public release. Distribution of this document is unlimited, and is not controlled by NATO policies or security regulations.	
13. Keywords/Descriptors: FORMAL SPECIFICATION, REQUIREMENTS, SYSTEM DEVELOPMENT	

14. Abstract:

This paper describes recent work at the SEI comparing three formal methods. We specified an example avionics problem using each method and used these specifications to evaluate the methods. The creation of the specification enabled us to understand, expose gaps in, and clarify ambiguities in the given requirements. We conclude with the results of our evaluation of the methods and reasons why we consider the formal restatement of the requirements to be an important part of system development.

UNCLASSIFIED/UNLIMITED

A.5.1 AC/243(Panel 11)TP/1

FORMAL SPECIFICATION AND REQUIREMENTS

Patrick R.H. Place* William G. Wood*

April 2, 1990

*Members of Technichal Staff at Software Engineering Institute (Sponsored by the U.S. Department of Defense). Carnegie Mellon University, Pittsburgh, PA 15213, USA.

Abstract

This paper describes recent work at the SEI comparing three formal methods. We specified an example avionics problem using each method and used these specifications to evaluate the methods. The creation of the specification enabled us to understand, expose gaps in, and clarify ambiguities in the given requirements. We conclude with the results of our evaluation of the methods and reasons why we consider the formal restatement of the requirements to be an important part of system development.

1. INTRODUCTION

One of the major problems with software intensive systems is the inadequacy of the system and software specification. The requirement documents usually define the major functions of the system or software adequately, but many of the details which should be drawn out, cleared up, and solidified in a more detailed specification are never seriously addressed, leading to flaws in the later implementation stages. The cost of fixing specification flaws detected at later stages in the life cycle is much greater than the cost of detecting and fixing them at the specification stage. Hence, it is important, as the first phase of development, to produce precise, complete, and consistent specifications of system operation from the requirements documents.

There are a number of techniques and languages that go by the term *formal methods*. These methods have a number of advantages over the methods currently used for system specification. The advantages of formal methods are: the precise definition of details at an appropriate level of abstraction, the ability to reason about properties of the representations derived, the ability to gain a detailed understanding of the operation of the system, and the ability to refine these specifications to lower levels and verify that the refined representation satisfies the specification.

This paper describes our recent work at the Software Engineering Institute in the use of formal methods to specify systems. We provide some background discussion in the use of formal methods to specify systems, discussion of our comparison and evaluation of three formal methods, and the results of our evaluations. We then discuss the problem requirements in light of the formal specifications; this discussion comments on ambiguities in the original requirements and the ways in which minor changes to the requirements could lead to a simpler system.

2. FORMAL METHODS BACKGROUND

Historically, mathematics has been the basis for modeling physical and conceptual systems. The use of mathematics increases our understanding of these systems, provides a mechanism for unambiguously communicating ideas about systems, and allows us to predict their behaviour using mathematical models. The predictive capability allows an engineer the luxury of finding and correcting faults in a model without building a full scale product. The mathematics for modeling the behaviour of software objects is still primitive, and work needs to be done within this field. Formal methods are firmly based on well known branches of mathematics, such as set theory, functions, algebra, various logics, and process algebra, and which have been used to model system behaviour. There are formal methods covering all aspects of software development, but our interest is in formal specification methods. A review of various formal methods, including

a description of where each method fits in the life cycle, has already been done by the Software Productivity Consortium [6].

Formal specification methods are generally classified as being appropriate to the specification of either sequential or concurrent systems.

Sequential systems generally terminate and perform computations where the output is determined by the input at the start of the computation. Examples of sequential methods are: Z [14], which was developed by Abrial and others at the Programming Research Group (PRG) at Oxford University: the Vienna development method (VDM) [5], which was developed at the IBM research labs in Vienna by Bjorner, Jones, and others; and Larch [3], which was developed at MIT. A comparison of these and many other sequential formal methods and their characteristics has been made by Sannella [13].

Concurrent systems are generally characterised by computations that run continuously, and are dependent on interactions with other computations and with changes in external conditions. Examples of concurrent methods are: communicating sequential processes (CSP) [4], which was developed by Hoare at PRG in Oxford: temporal logic [12], which was first proposed as a method for reasoning about concurrent programs by Manna and Pnueli at the Weizmann Institute of Science: and the calculus of communicating systems (CCS) [7], which was developed at Edinburgh University by Milner.

As with all engineering decisions, applying formal methods to a problem depends on the benefits of using formal methods over the benefits of using other methods. Formal methods will be used not because some zealots believe that they are the latest "silver bullet" of the software world, but because they provide distinct advantages over using other methods, and commercial pressures cause their usage. A general discussion comparing formal and traditional methods and their application to avionics systems has been presented by Wood [16]. The present paper concentrates on the use of formal methods as a means of restating the system requirements in a clear, concise, precise and consistent manner and the advantages of such a restatement.

Developing systems using formal methods provides confidence that the system implementation satisfies its requirements.

(1) The software developer is forced to be more rigorous in specification of the requirements, and the customer is therefore obliged to consider more details of operation early in the life cycle. Hence, formal software development starts from a more *complete* basis.

(2) The specification produced by these methods can be demonstrated to be *consistent*, since the methods support mathematical manipulations to prove or disprove consistency between parts. Both completeness and consistency can be demonstrated early in the life cycle.

(3) The specification serves as a communication between the specification stage, describing what the system is to do, and the design stage, describing how it is to be implemented.

(4) Implementations can be proved to be *correct* against the specifications.

We are not, naively, stating that formal software development does not have problems. Most of the tools supporting formal methods are research prototypes, and demonstrating consistency and correctness can only be done on a small scale. Software engineers and managers are

suspicious of specifications using "funny mathematical notations", and are not yet comfortable with a declarative style of representation, since most of their professional experience is with procedural languages. However, we consider the benefits of using formal methods to outweigh these problems.

Systems with high cost-of-failure characteristics require a higher assurance of safety and integrity than can be confidently achieved using non-formal techniques. If poor software can cause life-threatening accidents or substantial financial losses, or can yield unauthorised access to private information, or can cause loss of services at critical times, then the software should be specified and implemented in an extremely careful manner. Formal methods represent the most careful software development methods currently known. Hence formal methods are most applicable to systems with a high cost of failure, and are starting to be used in such a manner.

3. OUTLINE OF APPROACH TO FORMAL METHODS EVALUATION

The experiences reported in this paper were gained through the evaluation and comparison of three formal methods. This section outlines the work performed. We visited method developers and users to discover examples of the practical application of formal methods. Before we started evaluating the methods, we developed evaluation criteria and selected a problem and the methods to be investigated. We specified the problem using each of the methods, evaluated the results, and compared the applicability of the different methods.

3.1 Practical Use

We visited method developers to discover new developments in their techniques and to understand their perception of the state of the practice of formalisms. We visited method users to discover the nature and size of projects on which they were using formalisms and the acceptability of the formalisms to their companies. The following is a brief summary of the conclusions we made during the trips. A more complete summary of our visits is currently being prepared [10].

The existence of the UK Ministry of Defence draft standard, MOD 00-55 [8], which requires the use of formal methods in the development of safety critical systems, will have an effect on the software development community, particularly in the UK. Although MOD 00-55 in its present form may not be an enforceable standard, its existence will ensure that contractors and researchers will start experiments in the formal development of large scale systems. In the longer run, we expect MOD 00-55 to be the basis of an enforceable, and generally acceptable, standard for the defence contracting community requiring some level of formality in software development.

Interest in the application of formal methods to a wide range of systems is increasing; there is a growing number of examples of formal specifications and some companies are basing their business upon their ability to formally specify systems.

The belief that formal methods are hard to teach and use is now being shown to be false. Some universities are introducing formal methods at the very start of the undergraduate cur-

riculum. There are examples of staff from commercial development, rather than research, departments successfully applying formal methods as part of their standard system development practices.

3.2 Evaluation Criteria

We developed classification criteria in order to objectively compare different formal methods. The creation of these criteria was based on work in the classification and evaluation of methods and tools applicable to the entire system development life cycle [2, 15]. This work, however, covered more of the system development life cycle than appropriate for our investigations and was insufficiently detailed for the classification of specification techniques. Therefore, we took the three categories applicable to specification and refined them with appropriate criteria. Our definitions of the categories, with some examples of the criteria follow.

(1) The *representation* category consists of the concepts of a system that a specification technique could be used to describe. It is not an exhaustive list of concepts, but rather is a list of concepts that we consider to be of particular importance. Examples of criteria in this category are the representation of time and the style of specification employed by the technique.

(2) The *derivation* category consists of the ways in which specifications may be created from other specifications. Examples of derivation techniques are refinement and decomposition.

(3) The *examination* category consists of the properties that we may wish to show are present (or absent) in the specification of the system. The types of examination that may be carried out may be limited by the choice of specification technique. For example, it is possible to specify a system using a given technique but it is not possible to examine the system for absence of deadlock. Since the specification is a model of the system, behaviours (or functions) of the specification are assumed to be exhibited in the system. The criteria in this category include examinations for properties such as safety and liveness of the system.

3.3 The Sample Problem

We selected the requirements for a generic avionics system as our sample problem. It should be noted that the system, although not real, is realistic and suits our purposes in that it is small enough to be specified in a short space of time, yet large enough to be more than just a "toy" problem.

Essentially, the requirements describe an avionics system consisting of a mission control computer (MCC) and a number of devices which provide data to the MCC or pilot displays. These devices include an air data computer, an inertial navigation system, and a radar. In the event that a device fails, the requirements state that the MCC will estimate values for the data based on known conditions and previous values of the data. Further, if a device fails the pilot must be informed through both displays. A further component of the problem is a waypoint manager which maintains a list of coordinates; this determines the route that the aircraft must follow.

This problem was suitable for our purposes in that it consists of a number of independently acting, cooperating subsystems — the devices, the waypoint manager, the displays, and the control functions of the MCC. It contains strict timing requirements such as the requirement that a device failure message must be displayed for at least two seconds on one of the pilot displays. The waypoint manager introduces a "software device" which has different characteristics to the physical devices, notably a concentration on the description of a data structure. The possibility that devices might fail introduces into the system a notion, albeit explicit, of reliability, as well as the notion of timeouts so as to avoid indefinite delays waiting for information from a failed device.

The original requirements contained more components; however, these were omitted as they did not exhibit characteristics other than those of the components described above.

3.4 Methods Evaluated

Our efforts to date have involved the specification of the avionics problem in three formal specification techniques: CSP, VDM with extensions, and temporal logic. There are many other currently popular techniques we could have investigated: we chose these three for initial investigation because they are quite different in approach to system specification and so could be used to test our classification and evaluation criteria. If we could not use our criteria to distinguish between these techniques, then we would have to re-examine the criteria.

CSP is an example of a *process algebra* and is based on the notion that a process is defined by the possible sequences of *observable events* in which it may participate. Each observable event is a synchronous communication between the process and some other process (which may be the environment). CSP may define a process either *constructively*, by means of an explicit model of the possible orders of events, or *declaratively*, by means of predicates on the possible sequences of events (known as *traces*). It is possible to define a process both declaratively and constructively and to prove the correspondence (assuming it exists) between the two definitions.

VDM is an example of a *model-oriented* specification technique and was originally developed for the specification of sequential systems. We have used extensions to VDM [9] which introduce a model of a history of events in which the system will participate so that we can model the concurrent aspects of our problem. These histories correspond to the CSP notion of traces. Specifications written in VDM generally consist of a *state* (a collection of typed variables and invariants — predicates describing the relationship between the state variables) and *operations* (or *functions*) which modify the state.

Temporal logic is a particular form of *modal logic* and is based on the notion that the operation of a system can be described as a sequence of states and associated events. Then, a temporal logic specification of a system consists of a number of assertions using temporal and predicate logic operators that constrain the allowable states of the system. The temporal operators permit the specifier to constrain the allowable orderings of events of the system. One of the advantages of the particular temporal logic we used [1] is that a model can be constructed in a state machine language against which all of the temporal formulae may be checked, thus providing the specifier with greater confidence in the consistency of the specification.

4. EVALUATIONS

The full details of the evaluations are presented in a forthcoming SEI technical report [11]; this section summarises the evaluations.

At first glance, the three techniques seem roughly equal. They each have strengths and weaknesses, though in different criteria, as can be seen by examining the evaluations. This appearance of equality arises from giving each of the criteria equal weight which may not be appropriate for all classes of system. We believe the avionics system, for example, to be typical of *reactive* systems (systems that maintain continuing interaction with the environment and are generally unlikely to terminate).

For example, one of the criteria in the representation category concerned details of the user presentation. However, when simply restating requirements on reactive systems, such detail would be premature. Detailed decisions of the user interface should be left to a later stage in the system development. In the case of the avionics system, we would have had to create appropriate displays since this information was not presented in the requirements document. Obviously, if the system being specified were a user interface system the user presentation specification would be paramount and the criteria would be weighted differently.

Another of the criteria in the representation category concerned the manner in which data structures could be described. However, in the avionics example, there was only one, relatively uncomplicated data structure (the waypoint manager). We consider this relative lack of dependence on complicated data structures to be typical of reactive systems. The description of data structures may not be as important a factor in the specification as the description of the interactions between the subsystems.

In the derivations category, we found that perhaps the most important of the criteria, in terms of constructing a specification, was the ability to compose pieces of specification to form a specification of larger systems. Again indicating that a weighting of the evaluation criteria according to the type of problem (and development approach) is needed.

Finally, with respect to examinations, our visits lead us to believe that the greatest gain from formal specifications at present is to be made by restating the requirements in such a way that they may be understood by the entire project team, rather than creating a specification with all of the properties that could be determined through the examinations we listed.

5. THE PROBLEM REQUIREMENTS REVISITED

One interesting and unexpected (though not harmful) behaviour was the discovery that the INS and ADC devices could be treated as having identical behaviours. Using the specification techniques to create an abstract specification of the system makes this similarity in behaviour obvious. Such a clear statement of common behaviour has benefits for implementation where communication between the MCC and, say, the INS may reuse the code implementing communication between the MCC and the ADC. In fact, the same communication mechanism could also be used between the MCC and the radar. For the purpose of developing our specifications, we have assumed that all of the devices work in some polled manner. This may have been an

incorrect decision on our part. Equally well, the devices could operate by placing somewhere into memory the required information and potentially reporting their own status. This would fundamentally change the specification. Were we developing a real system, we would expect to query the system architects to discover how the devices actually work before progressing very far in writing the specification.

Another decision we made was to assume that we could use as many timers, of varying lengths as we need. Again, were we developing a real system specification, we would have to query the system architects concerning the nature of clock interrupts and the granularity of the timers that could be derived. It might be the case that the only interrupt available would be a simple, cyclically generated pulse. This would fundamentally alter the structure of our specification.

The second of these decisions may be thought of as a specification convenience, in that it is possible to model all of these independent clocks using a single clock process that generates interrupts to appropriate processes at the desired intervals. However, the issue of the manner in which we have assumed that the devices communicate information is a more fundamental issue. Our specification assumes that the devices have to be polled before they provide information; an alternative form of communication is one where the device continually updates an information store read by the MCC when it needs the information. Such a device behaviour would alter (perhaps simplify) the structure of the specification, and we would not expect it to be modeled by the behaviour described in our specifications. In both cases, the act of creating the specifications exposed the gap in the requirements document early rather than late in the development of the system.

The requirements referred to *the pilot and autopilot steering commands*; however, it is not clear whether these commands are given to, or accepted from, the pilot and autopilot by the rest of the system. We decided that these were commands sent to the pilot and autopilot. The use of formal specification exposed the ambiguity. Different readers of the requirements made the different interpretations. This difference was made visible when the formal specifications were compared.

The requirements stated that *the radar display must be updated every 200 msec.*. We used the formal specification to explore three possible alternatives. On the assumption that the requirements meant that the display had to be updated precisely every 200 ms., the most complicated specification was the one that most closely satisfied the requirements. However, this specification required the use of two timers, which would make a resultant conforming system more complicated, place a higher burden on interrupts, and would therefore be harder to test (or verify) for correctness. We could offer the system architects the alternative specifications, pointing out the consequences of each of the alternatives, and an appropriate change in the requirements could simplify the specification and resultant system, potentially leading to a cheaper, more reliable system. Using the specifications as models, we could predict the characteristics of the systems and determine which behaviour was most appropriate for the avionics system.

6. CONCLUSIONS

Our experience with the sample problem gives us confidence that use of an appropriate formal specification technique clarifies issues at the specification phase of a system's development and that formal specification techniques are applicable to the domain of avionics systems. Indeed, based on our work and the result of the survey of practitioners, we believe that formal specification techniques are ready for application to real systems.

Formal specifications provide a good basis for communication between team members. When we were comparing our specifications, it was easy to understand exactly what other authors had written due to the precise nature of the descriptions. Thus, we were quickly able to see differences between the different specifications and to remove these differences.

There is a growing interest in the use of formal specification, especially in Europe. To date, formalists in the USA have concentrated on verifying pieces of systems, though there is a growing interest in the use of formalism for specification.

That the classifications of the three techniques presented in this report showed differences between the techniques gave us some confidence in our classification criteria as a useful guide to distinguishing between specification techniques.

There are differences between specification techniques. The choice of formalism will affect the structure of the specification, which will in turn probably bias the program developed from the specification. Thus, the initial choice of specification technique is a very important step in the development of a system.

Based on our specifications, we would choose CSP for the specification of reactive systems. However, we do not consider that CSP is so much better than temporal logic that developers already versed in temporal logic should switch to CSP. We did not consider VDM to be particularly well suited to the specification of our avionics problem, though the VDM evaluation may have been adversely affected by the VDM specification following the CSP specification too closely. A second issue is that we have not considered the process of constructing a design from the specification. It is unclear how to construct a system design from a CSP specification, whereas the use of state machine models and temporal logic provide a more familiar method for the construction of a design. So although we prefer CSP for the construction and manipulation of a specification, considering the entire development life cycle, temporal logic may prove to be a more worthwhile approach.

Finally, the restatement of the requirements helped us to understand the avionics system and to prepare a description of the system that is precise and consistent. The creation of the formal specification helped us to identify commonalities in components, to expose gaps in the requirements, and to explore alternate possibilities for system components. The formal specification is consistent with our intuition concerning the avionics system and is a suitable starting place for the design process.

References

[1] E. M. Clarke and O. Grümberg. Research on Automatic Verification of Finite State Concurrent Systems. *Annual Review of Computer Science*, pages 269–289, 1987.

[2] R. Firth, W. G. Wood, R. Pethia, L. Roberts, V. Mosley, and T. Dolce. A Classification Scheme for Software Development Methods. Technical Report CMU/SEI-87-TR-41; DTIC: ADA 200606. Software Engineering Institute, November 1987.

[3] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in Five Easy Pieces. Technical report, Digital Systems Research Center, July 1985.

[4] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[5] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 1986.

[6] J. A. N. Lee and K. A. Nyberg. Strategies for Introducing Formal Methods into the Ada Life Cycle. Technical Report SPC-TR-88-002, Software Productivity Consortium, January 1988.

[7] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[8] MOD-00-55. Requirements for the Procurement of Safety Critical Software in Defence Equipment, 1989.

[9] J. Pedersen and M. Klein. Using the Vienna Development Method (VDM) to Formalize a Communication Protocol. Technical Report CMU/SEI-88-TR-26; DTIC: ADA 204750. Software Engineering Institute, November 1988.

[10] P. R. H. Place and W. G. Wood. Formal Specification Methods in Practice. In preparation for SEI Annual Technical Review.

[11] P. R. H. Place and W. G. Wood. Survey of Formal Specification Techniques for Reactive Systems. In preparation as an SEI technical report, 1990.

[12] A. Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. In *Current Trends in Concurrency*, pages 510–584. Springer-Verlag, 1985.

[13] D. Sannella. A Survey of Formal Software Development Methods. Technical Report ECS-LFCS-88-56, Edinburgh University, July 1988.

[14] J.M. Spivey. *The Z Notation*. Prentice-Hall, 1989.

[15] W. G. Wood, R. Pethia, L. Roberts Gold, and R. Firtn. A Guide to the Assessment of Software Development Methods. Technical Report CMU/SEI-88-TR-8; DTIC: ADA 197416. Software Engineering Institute, April 1988.

[16] W. G. Wood, D. P. Wood, P. R. H. Place, and D. W. McKeehan. Avionics System/Software Requirements Specification. In *Tenth Annual IEEE/AESS Dayton Chapter Symposium*, 1989.

NATO UNCLASSIFIED

REPORT DOCUMENTATION PAGE		
1. Recipient's Reference:	2. Further Reference:	
3. Originator's Reference: AC/243(Panel 11)TP/1	4. Security Classification: UNCLASSIFIED/UNLIMITED	
	5. Date: 15.04.91	6. Total Pages: 19
7. Title (NU): The "Spiral Model", some problems, many solutions?		
8. Presented at: AC/243(Panel 11) Symposium on Military Information Systems Engineering - RSRE, Malvern, UK - 8-10 May 1990		
9. Author's/Editor's: Dick W. Fikkert		
10. Author(s)/Editor(s) Address: TNO Physics and Electronics Laboratory P.O. Box 96864 2509 JG The Hague The Netherlands	11. NATO Staff Point of Contact: Defence Research Section NATO Headquarters B-1110 Brussels Belgium (Not a Distribution Centre)	
12. Distribution Statement: Approved for public release. Distribution of this document is unlimited, and is not controlled by NATO policies or security regulations.		
13. Keywords/Descriptors: SPIRAL MODEL, LIFE CYCLE MODELS, IT PROJECT MANAGEMENT, IT DEVELOPMENT APPROACHES, PROCUREMENT, QUALITY MANAGEMENT, QUALITY ASSURANCE, TECHNICAL DEVELOPMENT, EVOLUTIONARY PROCUREMENT, PROTOTYPING		

14. Abstract:

The author defends an alternative life cycle model called the Spiral Model introduced in literature by Dr. B.W. Boehm. This model is a generalization of other existing models and creates a risk-driven approach for the development of software intensive systems.

The paper emphasizes on finding additional rationale in favour of the Spiral Model. The author found some rationale by applying literature results to the spiral model and some by analyzing current approaches on problems that occurred when author applied and promoted the use of the Spiral Model.

The paper states that system development is made up of 5 domains (Life Cycle, Project Management, Procurement, Quality Management and Quality Assurance, Technical Development). These domains should be made less dependant of each other, otherwise alternative approaches like the Spiral Model or evolutionary procurement cannot be introduced.

The "Spiral Model", some problems, many solutions?

Dick W. Fikkert[1]
TNO Physics and Electronics Laboratory

Table of Contents:

---

1 TNO Physics and Electronics Laboratory
P.O. Box 96864
2509 JG Den Haag, The Netherlands
tel +31-70-3264 221
DFikkert@ccint1.MOD.UK (Internet)

1. OVERVIEW, OBJECTIVES AND RECOMMENDATIONS

Many are using an ASAP approach (Analyze, Specify and Produce; better known as the waterfall model) to manage their acquisition process. Therefore contractor's project managers have to use a similar approach to be able to discuss their project plans with their customers. This in turn leaves limited freedom to the development approach, which thus has to be based on a waterfall type of approach also.

Because of the mentioned dependencies, it seems attractive to fit new approaches like prototyping into the currently used life cycle [1. NATO-Policy90]. However, that will not solve problems that are inherent to the Life Cycle Model used. Furthermore it is plausible that not all types of software based systems will benefit from applying the same approach.
It can be concluded that there is a need for a development approach that is flexible enough to handle the development of quite different classes of software based systems, yet is acceptable for approaches used in management and acquisition.

It is the view of the author that the Spiral Model [2. Boehm88] is a good candidate for such an alternative approach. It is an objective of this paper to explain and reason about aspects of the spiral model that are (meant to be) different compared to other approaches. Experience gained by author through promotion and (limited) use of that life cycle model, shows that the procurement side and contractor's management will have at least 2 major questions about any changes in either development or procurement approaches.

The first question is: "The spiral model sounds like a good idea and it apparently solves some problems but how do I manage it?"
The second question addresses a chicken and egg problem but deserves proper attention because of the consequences of replacing procurement and development approaches. The question is: "I do know that the current approach has some deficiencies but how do you know the suggested alternative will work with real life projects? It may solve some problems but it might introduce many others".

Both questions, at least to some extent, can be dealt with by starting to use the suggested approaches on an individual experimental scale. But that

route will take a very long time indeed before new approaches will be applied to the development of large software intensive systems.
This paper addresses these and other questions by analysing relevant aspects of approaches. The analysis shows that some "problems" of the spiral model are also present in the waterfall model. Yet the "problems" are recognized in the spiral model only. Actually it will be shown that these "topics" solve problems present in other models. It is fair to note that the solution offered is not generally acceptable yet. It is the view of the author that experimental use of the spiral model should be encouraged just as the exchange of experience with its usage.

Because the Spiral Model includes incremental, evolutionary development it can be a candidate for managing that type of development. Also it includes the "why and when" topic as (dynamic) risk management. One detailed recommendations in this paper is to investigate whether evolution can be applied to the development approaches themselves to allow introduction of the spiral model.

Replacing development and/or procurement approaches has consequences in other areas. RSG.01 "on Distributed System Design Methodology" of AC/243 Panel.11 introduced the notion of five domains[2] that make up system development. This paper recognizes a slightly evolved set of domains addressing the Life Cycle, the Management of Projects, the Procurement, the Quality Management and Assurance and the Technical Development. The recognition of the domains does make sense if they are as independent as possible.
Thus for example a quality standard [16. ISO900x] should be usable with the waterfall model as well as the spiral model. The draft guide-line [4. ISO-draft-N33] to ISO 9001 states explicitly that it is independent of a development model.
Yet it is the author's opinion that it should be investigated whether this guide-line allows the application of the spiral model before the successor to N33 becomes an international standard. Also it is likely worthwhile to investigate whether the concept of "separation of concerns" (orthogonality of domains) can be introduced in the implementation of the NIAG report "on the Development and Implementation of Software Intensive C^3I Systems". That report addresses acquisition as well [5. NIAG88].

The mutual dependency of the distinguished domains currently limits the application of alternate approaches. Therefore efforts should be directed so that the domains of Procurement, the Life Cycle, Project Management,

[2] Author is a member of RSG.01 that recognized the domains of Development, Management, Documentation and Quality Assurance. [3. RSG89].

Quality Management and Assurance, Technical Development, and relevant standards and guide-lines could be made less dependent to ease the introduction of alternative approaches.

2. INTRODUCTION

2.1 What will be addressed technically?

Technically this paper addresses the topics Life Cycle Models and Project Management for the development of software based systems. A life cycle model can be defined [3] as a structured, common view of compound activities that need to be performed to complete the software development and maintenance. Different life cycle models have been published and some are generalizations of a number of others [2. Boehm88 and to some extent 6. McDerm-Ripk84] Note that more detailed life cycle models may include the element "product" as well.

Project Management generally includes a life cycle model implicitly. In this paper the two are distinct and complementary. One task of project management is the management of the activities that comprise the (application of a) life cycle model. Software project management includes many other aspects [7. Boehm-Ross89 and 8. Wideman89] like communication with subcontractors, departments, procurement, sales and the end user, motivation of team members, consensing, negotiating, coordinating and socializing.

This paper narrows its view of Project Management to that part that is concerned with the control of completion time, money, quality, organization and information [9. Wij-Ren-Sto88]. The reason being that especially that part is dependant on and limits the acceptance of alternative life cycle models.

---

[3] This definition fits the objective of a software process model: to structure the development process by defining activities, products and relations (Koenig). This definition however includes the objective as well as the means.
Another description of the primary functions of a software process model is: to determine the order of the stages involved in software development and evolution and to establish transition criteria for progressing from one stage to the next (Boehm).

2.2 The relationship with evolutionary procurement

What is the relation of project management and life cycle models with evolutionary procurement? Quite simply stated, even the promising evolutionary procurement needs to be managed. As explained in a previous section, it then also needs a Life Cycle Model that probably needs to be a traditional one (see Introduction). *Although not at all recommended in this paper* one could envisage that each evolution of a software product that is being procured, itself be developed in a traditional way.
Another important relation to evolutionary procurement is the influence of procurement methods on development focussed methods as explained in the introduction. Replacing an approach somewhere in the chain will have great impact on the whole chain.

2.3 Why a specific general life cycle model?

The apparent contradiction in this section's title is also the answer to the question. The spiral model defended in this paper is a generalization of many other life cycle models. When it is applied it could unwind into an equivalent of the others. In practice it will unwind into an appropriate mix of the other models. An evidence of its generality is that people who address the spiral model often have almost contradictory views on what it comprises. Indeed the application of the spiral model can produce quite different life cycle activities depending on real project conditions (risks). Furthermore the model distinguishes process and product oriented activities.

3. THE SPIRAL MODEL, A FIRST GLANCE

Experience shows that the introduction of the spiral model has not lead to general acceptance, even though it was proposed by a celebrity like Dr. Barry Boehm. Although Boehm's introductory paper on the spiral model mentions some rationale for the model, people apparently believe that the changes the model proposes, introduce problems which seem not to be present in their current approaches. To explain these problems we have to investigate the spiral model in some detail.

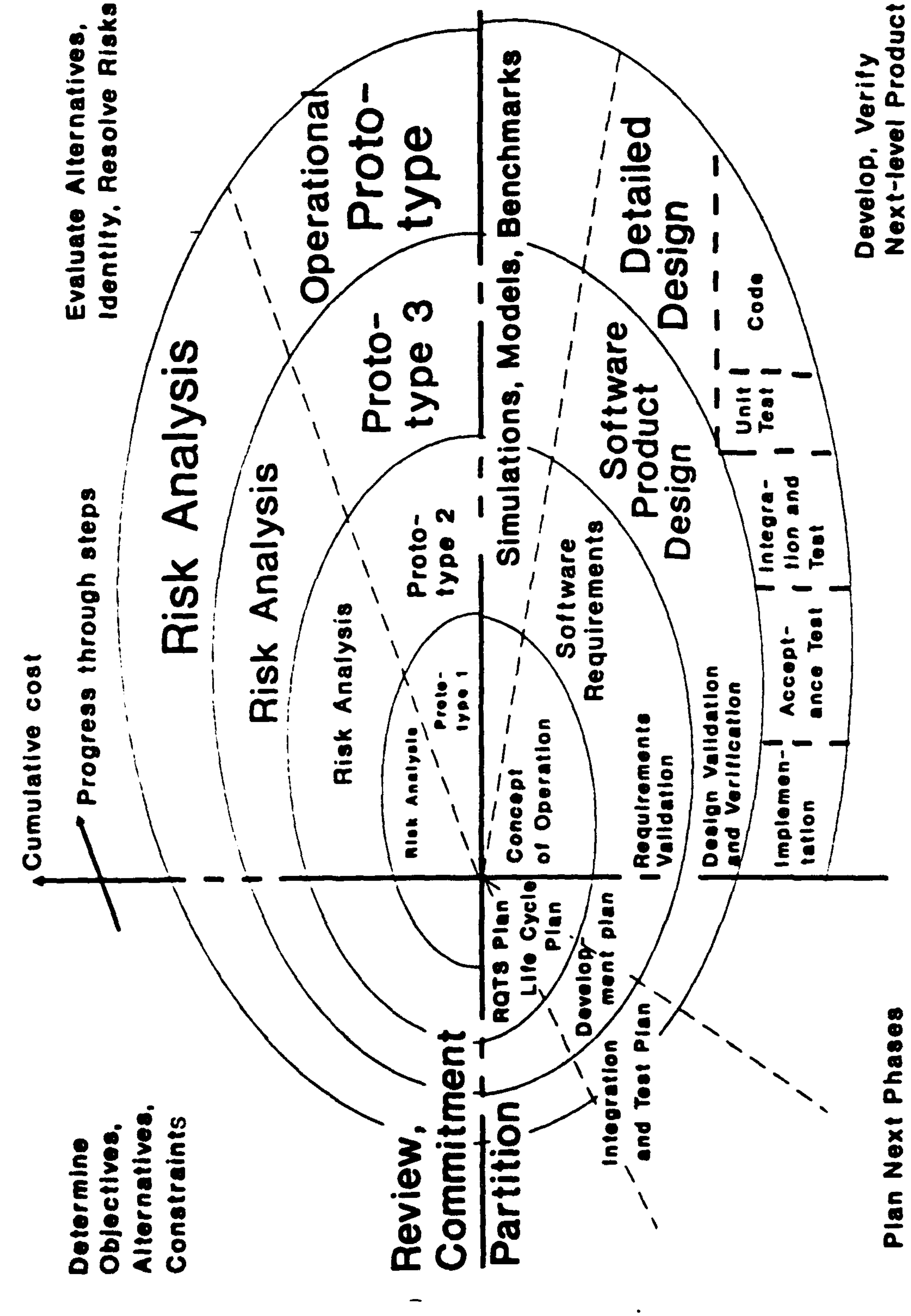


Figure 1. Spiral model of the software process.
(Copied with permission of Dr. Barry W. Boehm.)

3.1 The complete chart

Figure 1 is a chart of the spiral model. Although a picture is worth a thousand words, people usually need a few words of explanation with this one. Within the working group on Software Engineering of the Netherlands Society on Informatics we studied the original introductory paper [14. Boehm86] on the spiral model and still have some questions. One of our questions was "why is the spiral stretched out to the right?" This particular question was answered in an updated version of the paper published in IEEE computer [2. Boehm88]. It appears to be artistic license. If you try to draw the chart you will discover that either the letters are unreadable or the chart does not fit on the paper.

In a way the chart is the right one for a first glance because you will never forget it any more. For a first explanation I will ignore some elements. To explain the spiral model in words it is best to quote the IEEE version of Boehm's introductory paper:

> "The model reflects the underlying concept that
> each cycle involves a progression that
> addresses the same sequence of steps,
> for each portion of the product and
> for each of its levels of elaboration,
> from an overall concept of operation document
> down to coding of each individual program[4]"

Note that this description does not imply any ordering of for example successive portions and levels. The steps of each cycle are:

1. Determine objectives, alternative solutions and constraints.
2. Evaluate alternatives. Identify and resolve risks.
3. Develop, verify next-level product. (next-level is relative to the last completed cycle)
4. Plan next phase[5].

That's all that's needed to start thinking about the model and to relate it to some problems of traditional life cycle models and project management.

---

4 Note that some artistic licence has been taken with the layout of this compound sentence.

5 It's worthwhile to note that the first three steps are product oriented and risk driven. The last step is process oriented. If a risk driven approach is a necessity for the product steps, why shouldn't it be applied to the process as well? This idea has been introduced in [10. Boehm-Bels].

3.2 Another look

Another way of introducing the spiral model is to list some remarkable aspects of its application. In the next sections it will be apparent that its application:

- does not necessarily specify all requirements up front,
- does not necessarily execute activities in the same order compared to traditional approaches,
- does not necessarily specify all activities up front,
- does not necessarily at any point in time lead to a representation of the system at the same level of elaboration.

The words "not necessarily" should be understood as: whether it will or will not happen is determined during the execution of the project. The determination itself is based on risks that are determined during the project execution. More dynamic behaviour is hardly comprehensible!

Gone seems any basis for that part of project management that is concerned with the control of completion time, money, quality, organization and information. Let us investigate this questionable behaviour of the model.

4. SOME PROBLEMS (OR SOLUTIONS?)

All problems presented in this section have been encountered during the use and promotion of the spiral model by the author. However, some topics are of course covered in Boehm's introductory paper. The section on management and the short case are original. For all problems discussed, emphasis is directed to find additional rationale for the spiral model by applying results from literature to the spiral model. Also the results of the fruitful discussions in the working group "on Software Engineering"[6]. have been used for further rationale.

4.1 Not necessarily complete requirements

It is often stated that a moving target is bad for controlling development. Whom could say a similar thing? It could be argued that moving targets are bad for shooting!

6 This working group of the Netherlands Society on Informatics has studied the spiral model. The results have been presented by 3 members whom 1) Defended the waterfall model, 2) Analysed it's pitfalls and 3) Introduced the spiral model.

The application of the spiral model tends to *not* specify *all* details of all requirements up front. Since a full requirements document normally is one of the first milestones, control over the requirements phase and the final product seem to be lost.
It is the objective of this section to show that moving targets are sometimes, for certain classes of systems, inevitable. Software development and management should be able to cope with that fact[7]. The spiral model does not necessarily develop all requirements in full detail. That is a strong argument in favour of of that model, not a weakness.

4.1.1 Frozen requirements.

During a quality assurance seminar it was mentioned that "Working with a requirements document is like walking over water. It's easier when it's frozen." Yet not many people will realize that the water need not be completely frozen. For some, ice floes are enough. What is needed is a certain amount of frozen water. Furthermore the mechanism is frozen water, the objective is walking!
If you heard the above saying for the first time you may have appreciated it. More important however is whether there are some *fundamental reasons* why requirements often keep changing *other then* "the user does not know what he wants". The reason that requirements sometimes keep changing can be clarified if we study "software maintenance"

4.1.2 Maintenance types and costs

At least two types of maintenance can be distinguished.
- Corrective maintenance includes repair of hidden flaws of a new building and repair of a manufacturing fault in the steering column of a new car model.
- Adaptive maintenance includes restoring two rooms into one and mid life upgrades of equipment.

Not many, except IT-people, will call these activities maintenance. Yet that is what software maintenance is. Other types of non-IT

[7] Annex C (Definition of Evolutionary Acquisition) to [5. NIAG88] starts with "Evolutionary acquisition is an acquisition strategy which may be used to procure a system *expected to evolve during development* ...".

maintenance all have to do with wear and prevention of wear. Software does not suffer from wear [11. Lehman80]

It is generally accepted that:

- The ratio of the costs of software development over maintenance is 50/50 up to 30/70,
- 30% to 50% of the life cycle costs of software is spent on adaptive maintenance.

Having described some remarkable aspects of software maintenance, let us try to find an explanation for these phenomenons.

4.1.3 System types and maintenance

In 1980 Lehman distinguished three classes of software programs [11. Lehman80]. For each class the program's evolution (change in requirements) can be *predicted*. In practice systems are a mixture of the classes[8].

The second Lehman class is the easiest one to understand. That one includes systems which depend on for example laws, social security rules and strategies. Even if it was *formally proven* that the original software met the requirements, it has to be "maintained" as soon as the rules change. One could say that the requirements for systems belonging to this class are *time dependent*.

The third class consists of programs which are going to become part of the real world that these programs are trying to automate or support. These systems are almost impossible to specify completely. Large parts of C^3I-systems belong to third class. One could say that the requirements for systems belonging to this class depend on the system itself. In a sense these requirements are also time dependent because experimentation and real use will reveal new requirements.

The second and third type of systems are often characterized by saying that "the user does not know what he wants".

8 Lehman suggests to identify parts of a large program, that comply to his first class (S-class), during the partitioning process. Those system parts then can be implemented according to specifications that will not change (typical for S-class programs). Which parts can be handled that way is of course only known after the partitioning process. Not all work in all detail can be predicted up front' The spiral model also recognizes that fact.

4.1.4 The justification of inevitable change

It is a matter of simple extrapolation to understand that requirements will change during (especially long) developments too. Some requirements of a certain class of systems are time dependent. Would the world stop turning around during a long development cycle?

4.1.5 Change is inevitable

As explained in the previous sections, changing requirements are inevitable for certain commonly encountered classes of systems. It is useless to specify full details of requirements that will change during the development. The spiral model does not necessarily freeze or determine all requirements up front. By now it must be clear that it is a strong argument in favour of that model, it is not a weakness. How exactly the spiral model handles changing requirements is outside the scope of this paper[9].

As mentioned under "Maintenance types and costs", at least 50% of the "maintenance phase" are development activities. The spiral model recognizes this, names it software enhancements and incorporates these as a normal development cycle[10].

4.2 Activities are performed in the wrong order

When the abstract of this paper was prepared I had not written the paper, neither did I know the allocated time. After acceptance I heard that I could speak for 30 minutes on the tutorial day. Would I have known that in advance, my abstract would not cover "a second part of this paper" and probably would be more tutorial oriented.
Have things been done in the wrong order? That may be the wrong question. The point is that it is not reasonable to ask for the requirements "allocated time" and "session type" up front if you take the symposium organizer's job into account.

---

9 Basically the *strategy* to handle changing requirements will be determined depending on the risk involved to develop parts of the requirements. The strategy could lead to "develop a prototype" or "develop evolutionary", but other strategies like a combination of the two are possible too.

10 The spiral approach has the advantage that trade offs between alternative developments and traditional maintenance have to be made in step 1 and 2. If the money for enhancements and original development is supplied by almost independent sources [5. NIAG88], that problem has to be solved too.

The application of the spiral model tends to rearrange the order of activities compared to traditional approaches. It is possible that some design has started before all requirements are known. This could be (*but should not!*) interpreted as "we are back in the old days: code first and may be other activities will follow afterwards".

It feels right to develop all requirements of a system up front. However, if for example a COTS (Commercial off the Shelf) software package could be a real alternative in a particular situation, it is much better to do some evaluation of the commercial product right up front.
Boehm's introductory paper on the spiral model has an excellent introduction on "stage ordering problems"[11]. Its description of difficulties with evolutionary development are of particular interest to this conference. For example evolutions of a product may be very hard to accomplish if long range architectural and usage considerations have not been addressed or are hard to determine.

4.3 A short case

During the contract negotiations of an advanced tool for an experimental Programming Support Environment, two techniques to be supported by the tool were identified that were major risk items. While trying to minimize the risk at that stage, it was recognized that these techniques were not essential for the objective of the experimental PSE (it was the tool user that would benefit).
The support for the two techniques was offered as optional item (within the fixed time and budget constraints). This however was not acceptable because a specifications baseline should describe the tool exactly. Optional items could not be managed.

In this case the spiral model did identify and could resolve a major risk for the project. It would *tend* to use evolutionary development for the two techniques. Although in this case the specifications could be determined, the question was whether they could be implemented. This is clearly a case in which a lump sum contract is not the right approach[12].

---

11 [12. Boehm88] It covers the code-and-fix model, the stagewise and waterfall models, the evolutionary development model and the transform model.

12 An interesting table listing hardware contract approaches, advantages, requirements and problems is presented in [12. Griffiths89]. For lump sum contracts the following problem is described: "To the extent that requirements are not fully defined or when they are subject to variation, the employer puts the contractor at risk *and thus, probably himself*".

In the spiral approach the development of support for the two techniques would be "dormant" until knowledge about them would become a major risk for the rest of the project. Then some elaboration would be needed.
This is an example of the fact that system representations (design, detailed design, code etc.) need not be at the same level of elaboration across the whole system. It also is an example that evolutions sometimes have to be designed "a little". The development of future evolutions can not always totally be postponed until the next evolutionary version of a system.

4.4 It isn't manageable, is it?

In project management it is important that that activities are planned, committed to, executed and reviewed. *That certainly holds when applying the spiral model*. It is the intention of this section to reassess the type of control traditional project management techniques offer and show that the spiral model can improve control. However, the list of some remarkable aspects of the spiral model displayed in the section "another look" must worry some readers.
(Application of the model:

- does not necessarily specify all requirements up front,
- does not necessarily execute activities in the same, order compared to traditional approaches
- does not necessarily specify all activities up front,
- does not necessarily at any point in time lead to a representation of the system at the same level of elaboration.)

The author is convinced that currently the question of the management issue can only be dealt with through reasoning. Every time a question on the management of the spiral model is raised (to put it mildly), I have asked myself "how do we handle that problem currently?" It appears that some of us have forgotten what exactly project management techniques can control.

If project management (implicitly) defines a life cycle then the spiral model can not be managed because it has its own life cycle model. If the approach in project management must be the same as in other disciplines [8. Wideman89] then the spiral model violates some of the fundamentals of project management.
Still in development of software based systems others agree that "changes (to the acquisition procedures) must be made.to improve the chances of

success" [5. NIAG88] (in the author's view despite of project management techniques). So let us assume a change by separating the life cycle model from project management approaches. That is the first step to be able to use the spiral model.

Remember that the spiral model incorporates other models and can unfold as for example a waterfall model. If it is known that a particular project could well be tackled by a waterfall approach, it would be a good idea to start the project with a waterfall approach. *Then right after the development of the full plan, a switch to the spiral model could not harm at all*, because the risks identified would drive the spiral model into a waterfall model!
This seems an attractive way to experiment with the spiral model, especially if it appears that the waterfall is not that suited for the project.

4.4.1 Project management in general

Two important concepts to control projects are the control cycle and control elements. The control cycle consists of the steps execute, monitor, compare&forecast and adjust. The aspects of a project which can be controlled are the control elements completion time, money, quality, information and organization.
To be able to execute the control cycle it is necessary to have an actual state and a plan for each of the control elements. The step compare&forecast checks the actual situation with the planned one. The comparison can lead to an adjustment in the plans within the margins the plans contain for each of the control factors.

Just for reasoning , it will be assumed in the remaining of this section that the elements time, quality, information and organization are not adjustable. Money is the only element that can be adjusted. In other words: control over the project is possible only by spending more or less money.

(begin of assumption)

The assumption seems totally unreal. However, it is relevant and useful because:

- The assumption will clarify what exactly can be controlled in a project.
- The assumption is a reality as soon as the margins of the other controlling elements are exhausted.
- The assumption will clarify why margins in control elements can be exhausted

As long as the plan proves to be right the control cycle will not cause any adjustment. It must be obvious that the more advanced and unique the project is, the less accurate the estimates (the plan) can be. The control cycle however will perform its task because it uses the margins which ought to be present in any plan. The actual control of the project exists of the step adjust: it consumes the margin in the budget.
Note that the quantity of work to complete the project, given a certain approach is a constant[13], which is unknown at the start of the project.

If the margin contained in the plan is exhausted, the project is out of control in terms of project management. The step adjust is replaced by a step re-plan (a new cost estimate, negotiations etc.). This alternative step establishes feedback to the plan and corrects errors apparently present in the previous plan. The step re-plan should be an exception.

(end of assumption)

To summarize project control:

a) Project control consists in consuming the margins which ought to be present in the plan for each of the five control elements (completion time, money, quality, organization and information).
b) A project can not be controlled any more by project management techniques if the margins are exhausted. Someone has to "replenish stocks"

13 It would be nice if an estimate could influence for example the final quantity of work. In reality it may seem true however, most complex projects costs significantly more than the original estimate.

4.4.2 Why overrun?

Despite the application of project management techniques, complex projects overrun[14]. As explained in the previous section, control is only possible if a plan contains margins. Thus overrun is caused by insufficient margins.
Why does a plan contain these "errors". Many times, especially in smaller projects, there is no margin at all. As a consequence the step re-plan is a normal step in stead of an exception.

If margins were present and appeared to be to small, it is interesting to know how the margins were determined. Often it is an educated guess, supported by techniques and tools to calculate some "realistic" mean value for the project. On top of that commercial feasibility will adjust the estimates.
As already stated project control in not possible without margins. This implies that the control part of project management simply costs money. The customer pays if it is to large, the contractor pays if the margin is to small. An optimum can be realized if a re-plan step is considered to be normal. Effectively it then becomes a cost plus situation.
Note that the quantity of the work to execute the project, is independent of the final amount of margin left (positive or negative). It therefore seems beneficial to both the customer and the contractor to include the step re-plan in the normal project control cycle.

4.4.3 The spiral model, any improvement?

This spiral model has a plan-step as the last step of every cycle. The resulting plan and other results of the current cycle are the basis for review and commitment of all parties concerned. False expectations about the accuracy of the original plan are being anticipated because it is accepted that the step re-plan is normal, not exceptional.

The spiral model can not be a solution for the overrun of projects. In this model however, re-planning correctly will not be experienced as overrun. Overrun is inherent to the fact that its comparison base

14 Some interesting cost overruns as well as growing pre-project estimates (and some remedies) from other disciplines are listed in [12. Griffiths89].

is an *estimation*[15]. Even rough numbers are difficult to determine for unique complex projects.
It is often said that costs are not known in advance when applying the spiral model. That of course also holds for traditional approaches if the project plans had to be redone or the results of the project were not usable.
If accepted as an *estimate*, the result of any estimation method can be used with any approach, also with the spiral model. This model however seems to do a better job to control re-planning.

Is that all? Not really, but first we have to phrase the objective of project control: project performance. The spiral model is a real candidate to improve project performance.
It was already stated that the total amount of work to complete a project, given a certain approach, is a constant that is not known in advance. It is very likely that project performance could be improved by reordering the activities. That is what evolutionary and prototype approaches essentially are doing.
The spiral model uses risks determined *during* the execution of the project as a driving force to determine the order of activities. Therefore it probably will do a much better job to optimize the order of activities compared to any "upfront ordering" of other approaches.

5. THE SPIRAL MODEL, A SECOND CHANCE

5.1 Risks

Risk has not been addressed properly[16] in this paper so far. The spiral model addresses risks in two ways. First risks are identified through looking at alternative solutions. That is an approach that could be used in project management in general.
The spiral model uses risks resolution as the driving force to undertake activities. This has some non-trivial consequences. For example if a specific part of a design is a high risk item, that part should be

15 Estimate: to determine roughly the size, extent or nature of (Webster).
16 Boehm states that "The major distinguishing feature of the spiral model is that it creates a *risk-driven* approach to the software process rather than a primarily document-driven or code-driven process."

developed in much more detail then for example trivial parts (provided the high risk part is essential for the current release of the system).

It should be noted that risk analysis is to some extent already quite common. Using the risk analysis to determine the sequence of the tasks to undertake as well as the amount of elaboration within a task, is less common. Another difference is that the spiral model continuously assesses risks while other risk analysis approaches tend view risk as a static item. Often the risk analysis is done in the beginning to determine the price of the contract.

6. DETAILED RECOMMENDATIONS

This section also lists the general recommendations of the section "Introduction, objectives and recommendations".

- [2. Boehm88] "Even if an organization is not fully ready to adopt the entire spiral approach, there is one characteristic spiral model technique which can easily be adapted to *any* life cycle model, and which can provide many of the benefits of the spiral approach. This is the *Risk Management*[17] *Plan* ..."
- Experimental use of the spiral model and exchange of experience with its usage should be encouraged.
- The spiral model should be considered as a candidate life cycle model for evolutionary approaches.
- Effort should be directed to apply the evolutionary approach to the development approaches themselves to allow introduction of the spiral model in the cause of a project. It is of major help to answer questions about topics of the spiral model by analysing how and whether the current approaches address the questions.
- The domains of Procurement, the Life Cycle, Project Management, Quality Management and Assurance, Technical Development, and their relevant standards and guide-lines should be made more orthogonal to ease the introduction of alternative approaches.
- It is likely worthwhile to investigate whether "separation of concerns" (orthogonal domains) can be introduced in the implementation of the NIAG report "on the Development and Implementation of Software Intensive C^3I Systems".

[17] Risk management is probably (not yet read by author) also a main topic of [13. Risk89].

7. LIST OF REFERENCES

1. [NATO-Policy90] NATO policy guidance for the utilisation of simulation, prototyping, and testing in the development and acquisition of complex software based systems, AC/317 (WG/2)WP/55(Revised). NATO, Brussels, February 1990.
2. [Boehm88] Barry W. Boehm, A Spiral Model of Software Development and Enhancement. IEEE Computer May 1988, page 61-72
3. [RSG89] Final Report on Integrated Project Support Environments, AC/243(Panel 11/RSG.1)D/4. December 1989.
4. [ISO-draft-N33] Quality Systems- Guidelines for Quality Assurance (DRAFT). N33,ISO/TC176/SC2/WG5, February 1989.
5. [NIAG88] Report on the Development and Implementation of Software Intensive C^3I Systems, NIAG-D(88)17. NATO Brussels, June 1988
6. [McDerm-Ripk84] John McDermid and Knut Ripken, Life cycle support in the Ada environment. Cambridge University Press, ISBN 0 521 26042 6, 1984
7. [Boehm-Ross89] Barry W. Boehm and Rony Ross,Theory-W Software Project Management: Principles and Examples. IEEE Transactions on Software Engineering, vol 15, July 1989, page 902-916
8. [Wideman89] R. Max Wideman, Successful project control and execution (Keynote paper INTERNET 88). International Journal of Project Management,Vol 7 No 2 May 1989, page 109-113
9. [Wij-Ren-Sto88] Projectmatig werken. Gert Wijnen, Wilem Renes en Peter Storm, Marka Paperback, Het Spectrum bv, Utrecht 1988, ISBN 90 274 1923 X (in Netherlands)
10. [Boehm-Bels]Barry Boehm and Frank Bels, Applying Process Programming to the Spiral Model, probably SIGSOFT Software Engineering Notes 1989?
11. [Lehman80] Meir M. Lehman, Programs, Life Cycles, and Laws of Software Evolution. Proceedings of the IEEE, vol 68, no. 9, September 1980.
12. [Griffiths89] Frank Griffiths, Project contract strategy for 1992 and beyond. International Journal of Project Management, Vol 7 No 2 May 1989, page 69-83
13. [Risk89] Barry W. Boehm, IEEE Tutorial Volume on Software Risk Management, Catalogue Number EH291-5, 1989
14. [Boehm86] Barry W. Boehm, A Spiral Model of Software Development and Enhancement. *(ref 2. is an updated version)*. ACM SIGSOFT Software Engineering Notes, vol 11 no.4 August 1986, page 14-24

REPORT DOCUMENTATION PAGE

1. Recipient's Reference:	2. Further Reference:
3. Originator's Reference: AC/243(Panel 11)TP/1	4. Security Classification: UNCLASSIFIED/UNLIMITED 5. Date: 15.04.91 — 6. Total Pages: 10

7. Title (NU):

Automating the Development of Software

8. Presented at:

AC/243(Panel 11) Symposium on Military Information Systems Engineering - RSRE, Malvern, UK - 8-10 May 1990

9. Author's/Editor's:
Douglas R. Smith

10. Author(s)/Editor(s) Address:
Kestrel Institute
3260 Hillview Avenue
Palo Alto
California CA 94304
United States

11. NATO Staff Point of Contact:
Defence Research Section
NATO Headquarters
B-1110 Brussels
Belgium
(Not a Distribution Centre)

12. Distribution Statement:

Approved for public release. Distribution of this document is unlimited, and is not controlled by NATO policies or security regulations.

13. Keywords/Descriptors:

PROGRAMMING METHODOLOGY, FORMAL METHODS, AUTOMATED PROGRAMMING, REUSE, PROTOTYPING, EVOLUTION, ALGORITHM DESIGN, KNOWLEDGE-BASED TOOLS

14. Abstract:

This paper describes our work on automating the development of computer software. A model of the structure, design, and evolution of software systems is described. This model underlies KIDS (Kestrel Interactive Development System) which is used to interactively develop formal high-level specifications into correct and efficient programs. We also describe an extension of the model to handle the evolution of software systems.

AUTOMATING THE DEVELOPMENT OF SOFTWARE

Douglas R. Smith*

Contents

* Computer Scientist. Kestrel Institute. 3260 Hillview Avenue. Palo Alto. California 94304 USA

1 INTRODUCTION

This paper describes our work on automating the development of computer software. This work involves elaborating a model of the structure, design, and evolution of software systems. We believe that the following are essential characteristics of the next generation of software development tools.

- Formal specifications - to capture in a precise and implementation-independent notation the desired behavior of software;
- Semantics-preserving transformations - to base development and evolutionary activities on transformations that preserve some specified semantics:
- Formal design - to allow machine support for the design process: especially to allow the application of representations of general programming knowledge, domain-specific programming knowledge, and the capture and reuse of design decisions;
- Scalability - to uniformly model design and evolution processes at the program. module, and system levels;
- Evolution - to accomodate change as an integral part of design.

A key idea is that it is easier to understand, explain, build, and modify a complex object in increments rather than all at once. We describe our model of design (Section 2) and the KIDS system which embodies it (Section 3). In Section 4 we describe its extension to a model of software evolution. We conclude with a discussion of various features of the model and its possibilities for practical application.

2 MODEL OF SOFTWARE DESIGN

The diagram in Figure 1 presents the components and relationships of a simplified model of software design. The *domain model* is a formal representation of relevant aspects of the world within which the desired software is to be embedded. The *specification* component expresses constraints on the behavior of the desired software artifact. The *derivation structure* component is a record of the design decisions that connect the specifications to target code. Our model of design is a transformational one - the specifications are incrementally transformed in a stepwise refinement process into executable code that is provably consistent with the initial specifications. The final component, *code*, is a program specification expressed in a compilable programming language.

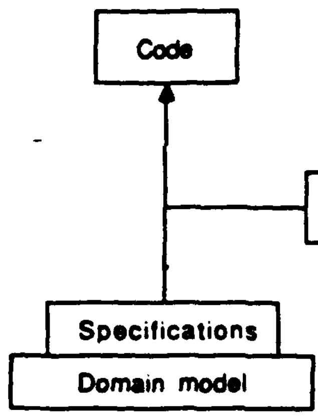


Figure 1: Design Structure

The data structure that comprises the four components in Figure 1 is called a *design structure*. The key constraint on a design structure is mathematical *consistency* between the components. That is, the specification is stated in terms of the underlying domain model and is consistent with its constraints. The derivation structure is a proof of consistency between the specification and code. The only assumptions used in deriving the code are those that are available in the model or in the specification itself.

2.1 Domain Model

If we want to specify and build a software system, then we need vocabulary and some expression of its semantics. *Domain models* express the objects, operations, relationships, agents, activities, and other assumptions and properties about the application domain. It is important to explicitly represent this information because *(i)* it is useful in achieving a common understanding among the developers of a system, *(ii)* it provides the vocabulary in which the requirements of the desired system are expressed, and *(iii)* the derivation of code from specifications is inference within the theory described by the domain model.

Domain models can be classified into two kinds: static and dynamic [8]. *Static models* are used in application domains that are essentially timeless. A database provides one example of a static model. It expresses the objects and relationships of a finite world. Static models are most generally expressed as theories in classical logic. *Dynamic models* on the other hand are used when the objects and relationships can change over time. They can be expressed by theories in temporal/modal logics or process models such as state transition diagrams [7].

2.2 Specifications

Specifications describe the intended behavior of a software system. They are expressed in terms of the vocabulary provided by the domain model. Specifications consist of formal interface descriptions (services provided and required input) plus constraints on allowable behavior of the desired software and the use of the system in context. The notion of a specification can be factored into functional, structural, performance, and environmental constraints:

- Functionality deals with the logical relations between inputs and outputs; that is, the interface with the rest of the software system's environment. Ideally the functional description is devoid of any structural constraints (implementation details). Functionality constraints describe what the system is intended to do.

- Structural constraints deal with the form of the software system: that is, how the system achieves its functional behavior. Structural constraints may describe the modules and abstract interfaces of a system, specify the use of routines from a standard library (rather than synthesizing similar code), specify the use of a certain communication protocol, etc. A LISP or ADA program can be thought of as a purely structural specification of a system.

- Performance deals with the resource utilization of a concrete program. Typical performance issues are program termination (finite consumption of resources), the amount of running time and/or memory space consumed, number of processors used, communication costs, etc. A specification might state that the target program should optimize a given cost function involving various aspects of performance.

- Environmental constraints describe the context in which the system will be used. Assumptions might describe the sizes of typical inputs, a probability measure on inputs, the relative frequency of calls on the system's utilities, number of processors available and their characteristics, etc. This information is essential in assessing whether the target code achieves its performance constraints. Environmental constraints import information from the domain model into the specification.

2.3 Derivation Structures

Derivation structures record the design decisions that connect specifications to code. Specifications are refined incrementally via the application of transformation rules into executable code that is provably consistent with the initial specifications [2, 3]. The derivation structure can be used for documenting and explaining the design and for helping to guide the design process.

One goal of our current work is to develop an abstract data type (ADT) of derivations and to validate its generality by applying it to a diversity of known implementation steps and design processes. A derivation ADT will have mechanisms for sequential and parallel composition, alternation, and iteration of development steps [14]. For our purposes, mechanisms for abstraction, application, and exception-handling will be vital to capturing general design processes found in KIDS [10, 11, 12]. The idea is that *ground derivations* can express design histories - a trace of the decisions made (by man or machine) during a derivation, and that *parameterized derivations* express design tactics - reusable methods for deriving code from specifications. An ADT for derivations would have many of the characteristics of a metalanguage, such as ML [6], since a derivation can be viewed as a metaprogram applied to a specification to derive code.

3 KIDS

KIDS (Kestrel Interactive Development System) [13] is an experimental knowledge-based software development system that integrates a number of sources of programming knowledge. It is used to interactively develop formal high-level specifications into correct and efficient programs. Tools for performing algorithm design, deductive inference, program simplification, finite differencing optimizations, partial evaluation, data structure refinement, conventional compilation, and others are available to the program developer. The KIDS tools have the characteristics of

1. being fully automatic (except the algorithm design tactics which require some interaction at present),
2. performing a well-defined and large grain-size development step, and
3. being correctness-preserving.

KIDS should be viewed as a front-end to a conventional compiler. It serves to raise the conceptual level from which the user can obtain efficient code by applying automated tools. Since any subset of the tools can be applied to a particular problem, the use of KIDS blends seamlessly into current programming methodologies. KIDS is unique among systems of its kind for having been used to design, optimize, and refine dozens of programs. We believe that KIDS could be developed to the point that it becomes economical to use for routine programming.

A user of KIDS develops a formal specification into a program by interactively applying a sequence of high-level transformations. During development, the user views a partially implemented specification annotated with input assumptions, invariants, and output conditions. A mouse is used to select a transformation from a command menu

and to apply it to a subexpression of the specification (via mouse-sensitive syntax). From the user's point of view the system allows the user to make high-level design decisions like, "design a divide-and-conquer algorithm for that specification" or "simplify that expression in context". We hope that decisions at this level will be both intuitive to the user and be high-level enough that useful programs can be derived within a reasonable number of steps.

Currently, KIDS runs on Symbolics, SUN-4, and SPARC workstations. It is built on top of REFINE [1], a commercial knowledge-based programming environment [1]. The REFINE environment provides

- an object-attribute-style database that is used to represent software-related objects via annotated abstract syntax trees;
- grammar-based parser/unparsers that translate between text and abstract syntax;
- a very-high-level language (also called REFINE) and compiler. The language supports first-order logic, set-theoretic data types and operations, and transformation and pattern constructs that support the creation of rules. The compiler generates CommonLisp code (prototype variants of the compiler can also produce C and ADA code).

The KIDS system is almost entirely written in REFINE and all of its operations work on the annotated abstract syntax tree representation of specifications in the REFINE database. A key feature of the unparsers/pretty-printers is the option for mouse-sensitive syntax - the pretty printer sets up active regions on the screen so that by moving the mouse around, the system can compute the nearest subexpression in the text and highlight it.

The user typically goes through the following steps in using KIDS for program development.

1. *Develop a domain theory* - The user builds up a domain theory by defining appropriate types and functions. KIDS has a *theory development* subsystem that supports the automated derivation of distributive laws for given functions. (A simple example is the distribution of addition over multiplication in arithmetic). The user may also supply other laws that allow high-level reasoning about the defined functions. Our experience has been that distributive and monotonicity laws provide most of the laws that are needed to support design and optimization.
2. *Create a specification* - The user enters a specification stated in terms of the underlying domain theory.

[1] REFINE is a trademark of Reasoning Systems, Inc., Palo Alto, California.

3. *Apply an algorithm design tactic* – The user selects an algorithm design tactic from a menu and applies it to a specification. Currently KIDS has tactics for simple problem reduction [10], divide-and-conquer [10], global search [12], and local search [9].

4. *Apply optimizations* – The KIDS system allows the application of optimization techniques such as simplification, partial evaluation, finite differencing, and other transformations. Each of the optimization methods are fully automatic and, with the exception of context-dependent simplification (which is arbitrarily hard), take only a few seconds.

5. *Apply data type refinements* – The user can select implementations for the high-level data types in the program. Data type refinement rules carry out the details of constructing the implementation.

6. *Compile* – The resulting code is translated by a conventional compiler. In a sense, KIDS can be regarded as a front-end to a compiler.

The user is free to apply any subset of the KIDS operations in any order – the above sequence is typical of our experiments in algorithm design.

KIDS will likely be useful in the near-range as an algorithm designer's workbench. It currently works best in application domains which are well-understood and whose foundation is readily formalized. Applications areas have included scheduling, combinatorial design, sorting and searching, computational geometry, pattern matching, routing for VLSI, and linear programming. We have been developing a number of tools, such as system-wide constraint maintenance, that could help with larger-scale programming tasks. A medium-term goal at Kestrel Institute is to apply KIDS to its own development.

4 EVOLUTION STRUCTURES

Software typically evolves over time - programmers must continually adapt it to meet changing needs and changing environments. Thus a successful model of design must accomodate change and evolution as an integral part of the design process.

The notion of a design structure in Figure 1 provides the framework for thinking about the evolutionary process. Initially the designer creates a simple, but consistent design. Then the designer iteratively begins transforming the design structure until a satisfactory design state is reached. These transformations are accomplished by making small but meaningful changes to either the domain model (to improve its accuracy and precision), to the specifications (to more accurately reflect the desired behavior), or to the derivation structure (to make better implementation choices). These changes are then propagated throughout the design structure in order to reestablish consistency. So, for example, if

we add an exceptional case to the domain model, the design system should propagate the exception into the specifications and finally through the derivation structure to be reflected in the target code. Note the shift away from the current notion of "replay" of a derivation structure on a modified specification (a kind of design-by-analogy) to the more deductive notion of propagating changes through a structure and reestablishing consistency. In a satisfactory design state the model is sufficiently accurate, the specifications have been elaborated to the point that they reflect accurately the needs of the users of the target software, and the derivation produces correct code with acceptable performance characteristics.

There are various ways to build a simple initial design structure. The domain model may include simplifying assumptions such as infinite memory or infinite precision arithmetic, or unbounded rationality in agents. Several data types may be confounded. The specification too may be oversimplified - perhaps dealing only with normal-case behavior and a very restricted subset of the desired functionality. The derivation structure may reflect a simple implementation strategy that yields correct executable code, but without much efficiency. Or it could implement the specification on a nonexistant very-high-level architecture. The derivation structure for a simple design state nonetheless records a derivation of code that is consistent with the model and specifications.

This model of evolution is based on the pioneering work of Goldman [5] and Feather [4] who are concerned with the evolution of specifications prior to implementation. Our approach applies the incremental elaboration idea to the entire design process and puts it on a rigorous basis.

5 CONCLUDING REMARKS

To conclude, we briefly discuss how our approach to software development treats several important issues in software engineering:

- *Maintenance* - This term is often used to cover at least two distinct development activities: error correction and program enhancement. The use of correctness-preserving transformations to develop programs from specifications gives us a far higher degree of confidence in the consistency of program and specification. The automation of the development process allows us to quickly prototype and validate a specification in order to assess its consistency with our intuitive needs. Thus an automated transformational development system should provide considerable assistance in removing inconsistencies (errors) as a source of problems in software. Software enhancement (evolution) remains then as the main driver of change in future software systems. The ability to treat evolution as a central mode of software development will be crucial to an effective model of software design.

- *Reuse* - The KIDS system and our model emphasize reuse, not of code, but of application domain knowledge, general programming knowledge (of algorithms, data structures, optimization techniques, etc.), and design decisions. We believe that this kind of reuse will provide greater long-term leverage than libraries of code.

- *Prototyping* - The ability to quickly obtain an executable refinement of a specification is the key to prototyping and the ability to perform validation on specifications. KIDS (via REFINE) provides for the automatic selection of default implementation choices for the constructs of our specification language. If defaults do not lead to acceptable performance for validation purposes, then better design decisions may be interactively applied in KIDS. The evolution mechanisms should enable elaborative changes to the prototype specification to be propagated to the code, reusing the old design decisions to aid the reestablishment of consistency.

- *Integration of Paradigms* - Our design model is much more formal and requires a different kind of human involvement than current programming practice. Generally there seems to be a serious problem with mixing formal methods with existing software and design practice. The clean modular design that is encouraged by formal specification methods is often broken down by optimizations during implementation - efficiency often demands the sharing of partial results. One must recover the initial system structure and subsequent elaborations and design decisions in existing software such that the evolution techniques can be applied. Another approach would be to apply the new methodology to certain critical modules and subsystems through their lifecycle. A module for which consistency between code and specifications is critical would be a suitable candidate.

Acknowledgements

I would like to thank Mike Lowry for his comments on a draft of this paper. This research was supported in part by the Office of Naval Research under Contract N00014-87-K-0550, and the Air Force Office of Scientific Research under Contract F49620-88-C-0033.

References

[1] Abraido-Fandiño, L. An overview of *REFINE™* 2.0. In *Proceedings of the Second International Symposium on Knowledge Engineering* (Madrid, Spain, April 8–10, 1987).

[2] Balzer, R., Cheatham, T. E., and Green, C. Software technology in the 1990's: using a new paradigm. *IEEE Computer 16*, 11 (November 1983), 39–45.

[3] CIP System Group. *The Munich Project CIP, Lecture Notes in Computer Science, Vol. 292*. Springer-Verlag, Berlin, 1987.

[4] FEATHER, M. *Constructing Specifications by Combining Parallel Elaborations.* Tech. Rep. RS-88-216, USC/Information Sciences Institute, December 1987. To appear in *IEEE TSE.*

[5] GOLDMAN, N. M. Three dimensions of design development. In *Proceedings of the 1983 National Conference on Artificial Intelligence* (Washington, D.C., August 22-26, 1983), AAAI, pp. 130-133.

[6] GORDON, M. J., MILNER, A. J., AND WADSWORTH, C. P. *Edinburgh LCF: A Mechanised Logic of Computation.* Springer-Verlag, Berlin, 1979. Lecture Notes in Computer Science, Vol. 78.

[7] HAREL, D. Statecharts: a visual approach to complex systems. *Science of Computer Programming 8*, 3 (June 1987), 231-274.

[8] JACKSON, M. A. *System Development International Series in Computer Science.* Prentice-Hall. Englewood Cliffs, NJ, 1983.

[9] LOWRY, M. R. *Algorithm Synthesis Through Problem Reformulation.* PhD thesis. Computer Science Department, Stanford University, 1989.

[10] SMITH, D. R. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence 27*, 1 (September 1985), 43-96. (Reprinted in *Readings in Artificial Intelligence and Software Engineering*, C. Rich and R. Waters, Eds., Los Altos, CA, Morgan Kaufmann, 1986.).

[11] SMITH, D. R. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming 8*, 3 (June 1987), 213-229. (also Technical Report KES.U.85.2, Kestrel Institute, March 1985).

[12] SMITH, D. R. *Structure and Design of Global Search Algorithms.* to appear in *Acta Informatica.* (also Tech. Rep. KES.U.87.12, Kestrel Institute, November 1987).

[13] SMITH, D. R. KIDS - a semi-automatic program development system. to appear in *IEEE Transactions on Software Engineering special issue on Formal Methods*, September 1990.

[14] WILE, D. S. Program developments: formal explanations of implementations. *Communications of the ACM 26*, 11 (November 1983), 902-911.

NATO UNCLASSIFIED

REPORT DOCUMENTATION PAGE

1. Recipient's Reference:	2. Further Reference:
3. Originator's Reference: AC/243(Panel 11)TP/1	4. Security Classification: UNCLASSIFIED/UNLIMITED 5. Date: 15.04.91 — 6. Total Pages: 9

7. Title (NU):

QUEST: Quality of Expert Systems

8. Presented at:

AC/243(Panel 11) Symposium on Military Information Systems Engineering - RSRE, Malvern, UK - 8-10 May 1990

9. Author's/Editor's:
M. Perre

10. Author(s)/Editor(s) Address: TNO Physics and Electronics Laboratory P.O. Box 96864 2509 JG The Hague The Netherlands	11. NATO Staff Point of Contact: Defence Research Section NATO Headquarters B-1110 Brussels Belgium (Not a Distribution Centre)

12. Distribution Statement:

Approved for public release. Distribution of this document is unlimited, and is not controlled by NATO policies or security regulations.

13. Keywords/Descriptors:

ARTIFICIAL INTELLIGENCE, DATABASES, EXPERT SYSTEMS, QUALITY CONTROL, KNOWLEDGEBASES

14. Abstract: This paper contains a summary of the results of the technology project "QUEST: Quality of Expert Systems", carried out under commission of the Dutch Ministry of Defence, Director Defense Research and Development. Participants in the project are TNO Physics and Electronics Laboratory (FEL-TNO), University of Limburg (RL) and the Research Institute for Knowledge Systems (RIKS). After an analysis of the problems encountered in Expert Systems development, a quality framework is developed which views the quality problem from three perspectives: the quality of the development process, the quality of the specifications and the quality of the expert system as a product. In order to get a better grasp of the problem a number of methods and techniques, derived from conventional and artificial intelligence systems development, are reviewed. Secondly the conceptual similarities between databases and knowledgebases are stressed. The use of conventional specification methods, in particular Nijssens Information Analysis Methodology (NIAM), is considered. It is demonstrated that the integration of database theory and artificial intelligence signifies a step in the direction of a better quality control of expert systems.

QUEST:

Quality of Expert Systems

by

M. Perre MA*

TABLE OF CONTENTS

* TNO Physics and Electronics Laboratory
Information Technology Division
Command and Control Information Systems/Knowledge Based Systems Group
P.O. Box 96864
2509 JG The Hague
The Netherlands
Fax +31 70 328 09 61
Phone +31 70 326 42 21

1. IN QUEST OF QUALITY: INTRODUCTION

From december 1988 until january 1990 TNO Physics and Electronics Laboratory (FEL-TNO), in collaboration with the University of Limburg (RL) and the Research Institute for Knowledge Systems (RIKS), worked on a technology project named "QUEST: Quality of Expert Systems" [1]. QUEST was carried out under commission of the Dutch Ministry of Defence, Director Defence Research and Development.

A strong motivation for this research project is the fact that more and more conventional systems contain intelligent modules, without the assurance that these modules satisfy the same rigorous quality measures as the conventional ones do. In comparison with conventional software systems the quality of expert systems is viewed as not being very satisfactory. Some of the more problematical aspects are knowledge acquisition, testing, evaluation and the maintenance of the knowledgebase. As yet there is not much unanimity with regard to the ways in which these problems have to be tackled [2]. This is an objectionable state of affairs, especially when you are dealing with critical applications, e.g. proces control systems in industry or nuclear power plants.

The same argument is also valid for military Command, Control, Communications and Intelligence (C^3I) systems . A characteristic of these systems is that they consist of large databases with which the deployment of men and material is coordinated. Starting point of this discussion is the thesis that a knowledgebase can be viewed as a collection of facts which can be manipulated with intelligent rules. These rules are also stored as objects in the knowledgebase [3]. An added convenience of integrating an inference engine with such a database system is that facilities like integrity, concurrency, security, recovery and distribution can now be used inside what we call a knowledgebase system, or expert system [4]. The main theme of this paper is that the integration of database theory and artificial intelligence signifies a step in the direction of a better quality control of expert systems

2. DEVELOPING A QUALITY FRAMEWORK

It needs no argument that statements about quality of programming code can only be made when the concept "quality" has been defined and made measurable. Whether a piece of code has "a good quality" is difficult to establish; The absence of quality, on the other hand, is much more obvious. The purpose of this paragraph is to set up a framework in which the quality of expert systems can be captured. There are three important aspects regarding the quality control problem:

(1) Analysis of an object system resulting in specifications
(2) Development process of an expert system
(3) The expert system as a product.

The relation between these three aspects is reflected in the following formula [1]:

PRODUCT = f(DEVELOPMENT PROCESS(SPECIFICATIONS)).

Applying the development process on the specifications results in the product "expert system". Quality can be controlled in three ways:
(1) Validation and verification of the system specifications
(2) A structured development process
(3) Test and evaluation of the product.

Specifications are of great importance during the whole development cycle of the expert system, and many errors are only discovered after implementation. This is the reason why the quality of specifications gets special attention in this paper. In the validation process specifications are compared with the "reality", or object system. It is determined whether system specifications are in agreement with user needs and demands. In the verification process specifications are compared with the implementation.

The necessity of a structured development process has long been acknowledged, an example is System Development Methodology (SDM) [5]. Since 1987 there exists a version of SDM which is used primarily for expert system development: Structured Knowledge Engineering (SKE) [6]. In this methodology more specific expert system activities like knowledge acquisition are worked out in detail. Another promising approach is that of Weitzel and Kerschberg, who are also proponents of so called expert database systems[7].

During the different phases in the development process (parts of) the system has to be tested on functionality and accordance with user needs [8]. Testing means looking at the behaviour of the system when it is "feeded" with a carefully chosen collection of data (test cases). Evaluation of an expert system can be viewed as conducting experiments with (parts of) a system and comparing the generated "advise" with the solutions to problems given by human experts [9].

3. FROM DATABASE TO KNOWLEDGEBASE

3.1 Introduction

Relational database management systems (RDBMS) are primarily being used for administrative applications. Concepts like data independence, data integrity, controlled redundancy, security and privacy are also very important when you are dealing with knowledgebase management systems (KBMS). Other reasons to adhere to the relational model are its conceptual simplicity and the familiarity system developers have with it. Most software producers are more experienced in using ORACLE and INGRES than LISP, PROLOG or AI-development tools like KEE, ART or Knowledge Craft. Moreover it is not easy to become acquainted with such advanced tools. By using an RDBMS update anomalies and redundant storage can be prevented. It also offers a flexible growth path when operational concepts are changed or data structures are modified.

3.2 Conceptual Modelling

When a knowledgebase is viewed as a special kind of database, various facilities of DBMS's could be used in KBMS's. Examples are recovery (to restore a

knowledgebase after a calamity, fault or power failure), concurrency (simultaneous utilization of a knowledgebase by different users), distribution (physically distribute a knowledgebase over different locations), security (protect a knowledgebase against unauthorized usage) and integrity (guard against inconsistencies of the knowledgebase) [4]. Especially this last point enables a direct relation with analysis and design methods of databases. Consequently, there is a need to build a conceptual model of a knowledge domain. A conceptual model can be placed between the internal model of a knowledgebase (the way in which the relations are physically represented) and the external model (the way the user sees the system) [10].

The conceptual model has to be a complete and consistent representation of a knowledge domain in which a distinction is being made between a knowledge schema (definitions of all used facts and relations) and the actual knowledgebase. This distinction can also be seen as a separation of types and instances. Consistency and completeness can be maintained by means of constraints on the actual knowledgebase. A conceptual model has to be constructed with a development method that lays down explicitly the definition of facts and their interrelations. In an AI-methodology like KADS four levels of knowledge are being distinguished [11]: domain, inference, task and strategic levels. NIAM, Nijssens Information Analysis Methodology, and ExtendedNIAM, can be used to strucure these four levels of knowledge [10,12,13].

3.3 Knowledgebase management system architectures

Several proposals have been made with respect to the architecture of knowledgebase management systems (KBMS). Often a distinction is made between loosely-coupled and tightly-coupled KBMS's [14].

A loosely-coupled KBMS is an external database management system (DBMS) that is interfaced with a logic programming language: e.g. the Oracle DBMS coupled with the logic language Prolog. Prolog-rules in this configuration can activate queries on the database.

In a tightly-coupled KBMS there is no distinction between a database system and a logic language. This can be realized in two ways: Firstly a logic language can be extende with database facilities like integrity, concurrency, security, recovery and distribution. Secondly a DBMS can be extended with deductive (Prolog-like) facilities. An interesting example of this architecture is POSTGRES, a further development of the DBMS INGRES (Post Ingres) [15,16].

POSTGRES is a tightly-coupled KBMS developed at the University of Southern California, Berkeley. The main aims of the project are to uphold the relational model and to provide facilities for "active" databases and inference, including forward and backward reasoning. In many applications it is very convenient to use triggers and alerters. Triggers are small pieces Structured Query Language (SQL) program which can be activated when changes are being made in the database (e.g. insert, delete or update). Alerters are comparable with triggers, but are activated by time or date.

The most revolutionary aspect of POSTGRES is the use of rules and procedures as if they were plain data items. Nijssen views an expert system as a system that contains human expertise and consists of a collection of related facts [3]. These facts can be inserted by a user, or can be derived by the system itself via on the basis of other facts and inference rules. Rules in POSTGRES can perform forward and backward chaining. This can be achieved by "early" and "late" evaluation. In the case of early evaluation a change in a data item that is containde in a rule will directly lead to activation of this rule. In the case of late evaluation the change only becomes obvious when a user queries that particular data item [15].

When inference rules are used in a KBMS, it is possible to perform a run-time "computation" of a relation. In other words the system has at its disposal an intension of the application (definition of tables, mutual dependencies and inference rules) and when needed computes the extension (the actual facts in tables).

Beside the innovations already mentioned, POSTGRES also offers the opportunity to represent complex objects, e.g. semantic networks. Actions like "generalise" or "specify" can easily be executed.

4. DAMOCLES: DAMAGE MONITORING AND CONTROL EXPERT SYSTEM

4.1 Introduction

TNO Physics and Electronics Laboratory, in collaboration with the NBCD School of the Royal Netherlands Navy, is developing DAMOCLES, a Damage Monitoring and Control Expert System. The main purpose of the DAMOCLES project is the development of an expert system which supports the damage control (DC) officer aboard Standard frigates in maintaining the operational availability of the vessel by safeguarding it and its crew from the effects of weapons, collisions, extreme weather conditions and other calamities. Basically DC-management includes the classical command and control cycle: status maintenance, situation assessment, planning, tasking and evaluation. An important way of making the total DC-organisation more effective is to improve the quality of the decisionmaking process by providing automated decision aids to the DC-officer in addition to the information processing and presentation facilities already available. This applies especially to damage assessment and planning.

4.2 Damage control management

In case a calamity has occurred, the DC-officer has to collect and combine data from different sources (sensors, communication systems, orderlies etc.) in order to assess the situation. On the basis of this, the DC-officer plans actions and looks after the careful execution of these. There are a number of problems interfering with the decision process: complexity of the vessel, uncertain and incomplete information, time pressure and catastrophic effects of wrong decisions. The DC-officer can only carry out his duty when he has a lot of

experience with the vessel and the procedures which have to be executed. Artificial intelligence provides the tools and techniques to use effectively the knowledge about the vessel and to accept, combine and fuse the data from sensors and reports. Therefore DAMOCLES has at its disposal: knowledge of the spatial structure of the vessel, knowledge of the state of the vessel, knowledge of procedures to be followed and knowledge based on experience of DC-officers. The system is not only intended to be used aboard navy frigates but also in training surroundings at the NBCD School.

The DAMOCLES project started with a detailed task analysis of the DC-officer in which the following main task areas were identified: stability monitoring and the prevention and repression of fire and damage. The DAMOCLES system assists the DC-officer in these tasks, most notably the evaluation of the situation (system monitoring and diagnosis), determination of the measures to be taken (planning) and monitoring of the execution (plan-monitoring).

4.3 Integrating artificial intelligence and database technologies

The DAMOCLES system is represented in terms of the layer model as put forward in the Structured Knowledge Engineering Methodology [6]: procedural level (strategic and task level), inference level and domain level. However, knowledge sources (inference level) and notions on the domain level [11] can be represented much more accurately with NIAM (Nijssens Information Analysis Methodology) [10]. In this way the quality of extensive databases and knowledge sources is better guaranteed.

An obvious choice for the representation of a conceptual model of reality, represented with NIAM, is a relational database. This implies that the domain and inference level can be implemented in a relational database. The procedural level has to be represented preferably in a relational programming language: Prolog is a logical choice. One important aspect of an expert system is the communication with the user. In DAMOCLES a graphics interface has been used which can be manipulated with Prolog. In this way a conceptual uniformity between functional specifications, database, programming language and graphics interface comes into being.

The possibilities of a combination of a relational database and Prolog surpasses those of conventional AI engineering environments. The following extras can be envisaged:

(1) Check on the consistency of knowledge,
(2) Secure the knowledge against unauthorized usage,
(3) Distributed storage of knowledge,
(4) Multitasking and distributed processing,
(5) Recovery facilities and
(6) Availability of many development tools.

When simulations are being executed (planning task) it is also possible to store different system states in separate databases.

The interface between DAMOCLES and the DC-officer has been given much attention. At present a prototype of DAMOCLES is running on a Sun-workstation equipped with a colour graphic display in order to present images of underlying technical ship systems (e.g. ventilation and high pressure sea water). These images consist of coloured block diagrams containing all relevant information required for monitoring and control of these systems. In addition to this, a 3-dimensional image of the ship with overlays of technical systems is used to convey spatial information.

A relational database enhanced with Prolog offers good opportunities for the transparant development of extensive and highly qualitative expert systems [17]. The usefulness of artificial intelligence and expert systems in DC-management is demonstrated by the development process of the DAMOCLES system.

5. CONCLUSIONS

The quality framework presented at the beginning of this paper is a good guide-line for tackling the quality problem of expert systems. The concept quality is viewed from three perspectives: the system specifications, the structuring of the development process and the expert system as a product. The knowledgebase is identified as the most important part of an expert system. Inference mechanism, man-machine interface and explanation facilities contain such an amount of "conventional" components, that quality control can be achieved with well known methods and techniques. Reliability and maintainability of the knowledgebase are recognized as the principal quality criteria.

The development process of expert systems cannot easily be controlled. As opposed to "conventional" life-cycle methodologies, there is no accepted version for expert systems. Experience accumulated in the application of SKE and the Kerschberg-Weitzel model could eventually lead to an easing of the problem.

The extended version of (conventional) NIAM provides good opportunities for specifying knowledgebases. Especially the way in which completeness and consistency of the knowledge model is guaranteed, makes ENIAM a far better choice than many other AI-specification methods.

Evaluation and testing of expert systems is an underdeveloped field of study. Often it is not possible to test the advise given by the system against an objective standard. Methods focussed on a structured generation of expert system test cases are not yet available. This is why the use of "conventional" evaluation and test methodologies is advocated.

The example given of a system that is being developed along the lines presented in this paper shows that an integration of notions derived from database technology and artificial intelligence can be very helpful in the creation of a non-trivial expert system.

6. REFERENCES

[1] Lenting, J.H.J., M. Perre, "QUEST: Kwaliteit van Expertsystemen", FEL-90-A012, 1990.

[2] Napheys, B., D. Herkimer, "A Look at Loosely-Coupled Prolog Database Systems",In: Kerschberg, L. (ed.), "Expert Database Systems, Proceedings from the Second International Conference", 1989 , pp. 257-271.

[3] Nijssen, G.M., "Knowledge Engineering, Conceptual Schemas, SQL and Expert Systems: A Unifying Point of View", In: "Relationele Database Software, 5e Generatie Expertsystemen en Informatie-analyse", Congres-syllabus NOVI, 1986, pp. 1-38.

[4] Date, C.J., "An introduction to database systems: Volume II", Addison-Wesley, 1983.

[5] Turner, W.S. (et al.), "System development methodology", 1987.

[6] Structured Knowledge Engineering, Syllabus Bolesian Systems Europe B.V., 1988.

[7] Weitzel, J.R., L. Kerschberg, "Developing Knowledge-Based Systems: Reorganizing the System Development Life Cycle", In: "Communications of the ACM", Vol. 32, Nr. 4, 1989, pp. 482-488.

[8] Myers, G.J., "The Art of Software Testing", Wiley, 1979.

[9] Llinas, J., Rizzi, S., "The Test and Evaluation Process for Knowledge Based Systems", Technical Report F30602-85-C-0313, Calspan Corporation, June, 1987.

[10] Nijssen, G.M., Halpin, T.A., "Conceptual Schema and Relational Database Design: A Fact oriented Approach", Prentice Hall, 1989.

[11] Breuker, J. (ed.), "Model-Driven Knowledge Acquisition: Interpretation Models", University of Amsterdam, 1987.

[12] Creasy, P.N., "Extending Graphical Conceptual Schema Languages", University of Queensland, 1988.

[13] Creasy, P.N., "ENIAM: A More Complete Conceptual Schema Language", In: "Proceedings of the Fifteenth International Conference on Very Large Databases", 1989, pp. 107-114.

[14] Stonebraker, M., M. Hearst, "Future Trends in Expert Database Systems", In: Kerschberg, L. (ed.), "Expert Database Systems, Proceedings from the Second International Conference", 1989, pp. 3-20.

[15] Stonebraker, M. (et al.), "The POSTGRES Rule Manager", In: "IEEE Transactions on Software Engineering", Vol. 14, Nr. 7, Juli 1988, pp. 897-907.

[16] Stonebraker, M., L.A. Rowe, "The design of POSTGRES", In: "Proceedings of the ACM-SIGMOD Conference on Management of Data", 1986, pp. 340-355.

[17] Brodie, M.L. (et al.),"Future Artificial Intelligence Requirements for Intelligent Database Systems", In: Kerschberg, L. (ed.), "Expert Database Systems, Proceedings from the Second International Conference", 1989, pp. 45-62.

REPORT DOCUMENTATION PAGE

1. Recipient's Reference:	2. Further Reference:
3. Originator's Reference: AC/243(Panel 11)TP/1	4. Security Classification: UNCLASSIFIED/UNLIMITED 5. Date: 15.04.91 — 6. Total Pages: 9

7. Title (NU):

TEN15 - A High Integrity Kernel for Software Engineering Applications

8. Presented at:

AC/243(Panel 11) Symposium on Military Information Systems Engineering - RSRE, Malvern, UK - 8-10 May 1990

9. Author's/Editor's:
Dr. N.E. Peeling

10. Author(s)/Editor(s) Address:	11. NATO Staff Point of Contact:
RSRE MOD (PE) St. Andrews Road Malvern Worcestershire WR14 3PS United Kingdom	Defence Research Section NATO Headquarters B-1110 Brussels Belgium (Not a Distribution Centre)

12. Distribution Statement:

13. Keywords/Descriptors:

SOFTWARE INTERFACES, PORTABILITY, HIGH INTEGRITY KERNEL

14. Abstract:

The TEN15 project is developing interfaces that decouple software systems written in any high-level language from the hardware architectures on which these systems run. The lowest level interface, TEN15 Distribution Format (TDF), is a potential Architecture Neutral Distribution Format (ANDF) in which portable software can be distributed for a wide range of machines. Ten15 is strongly typed TDF with additional features to support system programming. Ten15 is a good high-integrity kernel for supporting high-functionality, portable secure systems.

TEN15 - A HIGH INTEGRITY KERNEL FOR SOFTWARE ENGINEERING APPLICATIONS

Dr N E Peeling

1 INTRODUCTION

2 SOFTWARE PROGRAMMING AND DISTRIBUTION INTERFACES

2.1 TDF

2.2 The Ten15 high-integrity kernel

3 RELATIONSHIP WITH PTIs

4 CURRENT STATUS AND FUTURE PLANS

TEN15 - A HIGH INTEGRITY KERNEL FOR SOFTWARE ENGINEERING APPLICATIONS

1 INTRODUCTION

Computing Division at RSRE are working on techniques to improve the functionality, performance and integrity of portable systems. The defence interest centres on the application of these techniques to the development of operational secure Command and Information (CIS) systems, and the development of secure project development environments for such systems.

RSRE are developing interfaces that decouple software systems written in any of a wide range of programming languages from the architectures on which those systems will run. These interfaces are in effect abstract machines because they completely hide the underlying machine.

Two interfaces have been developed, the lower-level interface is called TDF (Ten15 Distribution Format) and addresses the problems of code-generation. It can be thought of an extension of the idea of a universal compiler intermediate language. TDF is a potential ANDF (Architecture Neutral Distribution Format) which is a programming language independent, machine independent format for the distribution of shrink-wrapped software products.

On top of TDF there is a higher level interface called simply Ten15 [1], which can be thought of as TDF with a comprehensive, rigidly enforced type system. The purpose of the type system is to allow systems to be built on top of Ten15 which are both efficient and offer high-integrity without limiting functionality. The type system is also comprehensive enough to allow Ten15 to describe system programs that manipulate a permanent datastore, and resources over a LAN.

A Ten15 system requires two main components: a portable translator to TDF; and a run-time environment for accessing system resources such as main memory, filestore, networks and other peripherals. Rigidly enforcing the type system in Ten15 allows the Ten15 kernel to implement safe, fine-grained resource allocation. The Ten15 kernel can implement a heap-oriented, garbage-collected common address space, which in turn allows it to support first-class procedure values. The type system ensures that users' programs can coexist in a common memory without any danger of one user's mistakes affecting another user's program or data. These sorts of feature allow modern, highly interactive, object-oriented systems to be implemented efficiently and portably on top of the Ten15

kernel. Efficient, safe encapsulation of data using object-oriented techniques is one of the key characteristics needed to implement modern secure systems.

2 SOFTWARE PROGRAMMING AND DISTRIBUTION INTERFACES

The two interfaces, TDF and Ten15, serve different purposes:

2.1 TDF

As mentioned above, TDF is being developed as a potential Architecture Neutral Distribution Format (ANDF). An ANDF can be used to distribute shrink-wrapped software products. An Independent Software Vendor (ISV) will produce a single version of a product in ANDF which contains embedded system calls to resources in a system environment. Customers can then purchase a piece of shrink-wrapped, ANDF software which can be installed on any machine which provides the necessary system resources.

A widely accepted ANDF would help create as strong a market in software products for the open systems running on the range of architectures that support UNIX as currently exists within the compatible PC world. Such a market would benefit ISVs by both enlarging and opening up the market on UNIX, and reducing the costs of supporting versions on multiple architectures. An ANDF would benefit end-users by increasing software availability, providing a more competitive market in software which should drive down software prices as well as allowing a freer choice of hardware.

Although the development of a standard ANDF is being driven by the desire to create a more vigorous software products market for UNIX, a widely available ANDF standard would have a number of other uses, many of which are of interest to the defence community.

ANDF provides a interface that decouples programs from the features of any specific architecture. This includes systems that prior to the availability of an ANDF were not portable because they generated machine-code - e.g. Ada compilation systems. If a situation arises where most important software tools are targeted at ANDF then it becomes much easier to change the underlying hardware base, without affecting the software the user wants to run. Two obvious advantages to the defence community are, first, the potential to reduce the costs of mid-life hardware upgrades, and secondly, the freedom to easily introduce new defence-specific architectures e.g. secure computer hardware such as SMITE [2] or verified safety-critical processors such as VIPER [3].

An ANDF provides a single common point that all software passes through, no matter what programming language it is written in and no matter what machine it is targeted to run on. This means that an ANDF provides an opportunity to significantly improve interoperability, both in mixed programming language systems and in mixed hardware systems. For mixed language working, ANDF provides a common format in which conventions can be defined for the new generation of compilers for the high-level programming languages that produce ANDF output. These standard conventions can maximise the extent to which the different languages can interoperate. In a mixed hardware environment, actions to be performed elsewhere in the network can be described in ANDF which is then sent to the remote node in the knowledge that it can be translated to appropriate machine instructions on the remote node.

TDF's strengths are a result of the fact that it is pitched at a higher level than more conventional abstract stack or register intermediate languages (such as P-Code, or the gcc intermediate language). It is defined at the level of abstractions of programming language features - it contains abstractions of: numbers, variables, procedures, pointers, loops, conditionals, exceptions, signals, threads etc. The coverage of language features makes it suitable for conventional languages such as ANSI C, Pascal and Ada, and also for more advanced languages such as Lisp and ML.

The close relationship of TDF and Ten15 also means that if TDF becomes the industry standard ANDF then a secure Ten15-like interface can be very easily built on top of TDF which will be available on all machines for which there is TDF support. A high-integrity Ten15 kernel would be contructed from a high-level type checker which generates TDF, a standard TDF installer and the Ten15 run-time environment.

2.2 The Ten15 high-integrity kernel

The single most important defence application for Ten15 is its use as a high-integrity kernel for portable secure systems. Ten15 is not in itself a secure system because it does not define and impose any particular security model: rather it provides helpful implementation mechanisms that allows the designer of a secure system to design a system that mandates any particular security model.

The Secure Processor Research team at RSRE has determined that there are four essential mechanisms needed for the implementation of secure systems [4]:

1. Unforgeable, opaque addresses - which allow hidden objects to be created. The opacity of the addresses means that the creation of a hidden object can give the

creator no possible information about other users, as might happen if the addresses could be interpreted as physical addresses in memory.

2. Data Hiding - which allows control of access to sensitive data, by hiding data behind a procedural interface.

3. Pedigree - which guarantees the origins of critical objects, as they are passed around within a system.

4. Context - which guarantees the correct authentication of users accessing the system.

In most existing secure systems these mechanisms can only be provided by severely limiting the ways in which a system can be used. For example, pedigree can trivially be provided by forbidding critical objects being moved dynamically within the system. As a result secure systems are often extremely inflexible to the changing needs of the users of the system.

Ten15 can provide the four mechanisms is a very flexible manner, largely as a result of the powerful, rigidly enforced type system that permeates a Ten15 system. In Ten15, objects can be created, encapsulated, passed between mainstore processes, and between mainstore and filestore, in a totally unpredicted, but secure manner.

Ten15 is defined algebraically and is a basis for the formal analysis of system properties. This makes it a suitable input to static analysis tools for use in validating security or safety critical applications. Ten15 has two potential advantages over existing methods: first, it allows the possibility of analysing complete systems (as opposed to just free-standing programs); and secondly, it can be used to generate trusted code directly from the Ten15 which was analysed. Further research is planned with the University of York to study techniques for building trusted code generators for Ten15.

Because Ten15 can represent arbitrarily complex programs, existing static analysis techniques would only be able to analyse a subset of Ten15 programs. We hope that the presence of a range of compilers for high-level languages that generate Ten15 will encourage the development of increasingly advanced static analysis techniques which will eventually be able to tackle all of Ten15, thus allowing analysis of programs written in all the languages that compile to Ten15.

The role of TDF in aiding interoperability in a heterogeneous hardware environment was mentioned in the last section. Ten15 adds advanced features for dynamically creating remote procedure calls and a common view of a structured, high-integrity filestore, as further aids to interoperability.

The concept of filestore in Ten15 extends programming language data structuring in an obvious way onto the backing store. The approach adopted is to provide general purpose primitives which separate the idea of mainstore pointers from disc pointers. Bringing a data structure from disc into mainstore, i.e. turning a disc pointer into a mainstore pointer, has to be done explicitly. The mechanisms underlying the Ten15 filestore are capable of supporting complex databases where individual items can, if necessary, be very small (the so-called "fine-grain" database). The Ten15 filestore is a potentially more flexible and efficient implementation mechanism for building distributed secure databases than are existing technologies such as relational and entity/relationship/attribute databases, and will be much easier to integrate with the programming languages used to implement the system.

The provision of a sophisticated storage allocation mechanism was discussed briefly in the introduction. The implementation of a garbage-collected common address space provides a mechanism whereby mainstore datastructures can be efficiently communicated or shared between different programs. This makes Ten15 a very good kernel for developing high-productivity programming environments. Such environments are becoming increasingly important as software prototyping plays a larger part in the design process for large defence systems. Software prototyping enables requirements to be clearly formulated and design trade-offs to be assessed. Unlike other systems which have been used for prototyping but which depend on special purpose hardware (e.g. LISP, SMALLTALK and the RATIONAL machines) Ten15 can be a cost-effective basis for both the development and operational systems.

3 RELATIONSHIP WITH PTIs

Both TDF and Ten15 have been described as interfaces. Much work is currently in hand on developing interface models for PSEs and specifically this has led to the concept of Portable Tools Interfaces (PTIs). The TDF and Ten15 interfaces fulfil a rather different finction from PTIs as will now be explained.

Let us first determine what requirements PTIs such as PCTE+ [5] satisfy. PCTE+'s most important feature is its entity management system (EMS). Other features such as the process model and the user interface are to a certain extent being overtaken by developments at the UNIX system level. for example OSF/Motif is likely to become the defacto standard presentation manager. and the development of lightweight process threads in UNIX kernels such as Mach will probably find their way into the IEEE POSIX standard.

The EMS grew out of an appreciation that a UNIX-like filestore provided little or no support for the complex interactions between the different types of data-objects that must be handled in a large scale software development project. Such objects include software, changes logs, data files, test and evaluation files, documentation, PERT charts, other management planning aids, etc. etc. The EMS provides a sophisticated data repository for storing, modifying and browsing this project database. More efficient project management through the use of PSEs has been identified as a key requirement for reducing the costs of procuring large software systems. This explains why PCTE+ has attracted so much interest and support from both the civil and defence communities: from IEPG and from ECMA.

The EMS supports the sorts of objects/attributes/relationships required in project management. It does not however provide support for representing the sorts of datastructure that arise within programs and which need to be communicated between different software modules. For example, consider the problem of storing a complex piece of data from a software system in the EMS. Examples of the sort of data handled in sophisticated tools might be a table of data from a spreadsheet, or the output from a syntax analyser for a human readable language. The table from the spreadsheet is probably represented in its system as a large two dimensional array of data, where each column of the array holds a different class of data, and each row is an entry in the table. The output from a syntax analyser is likely to be a tree datastructure which gives the syntactic structure of the language. PCTE+ provides no support for large arrays, or for tree structures, so to store either of these data items in the EMS the user would have to explicitly "flatten"the datastructure into a linear stream of bytes in order to store it in the contents field of an object. To do this the tool writer would have to choose a convention for turning the two dimensional array or the tree structure into a linear stream of bytes. This convention (and there are an infinite number of acceptable conventions) would have to be understood by any other tool writer who wished to read the datastructures from the EMS.

The restriction on the datastructures that can be handled by the EMS means that PCTE+ provides little explicit support to software tool interoperability, configuration management of software modules, and permanent storage of program generated data. The EMS of PCTE+ and other PTIs do little to help manage the data generated by the activities within a software intensive project which are concerned with the actual design and implementation of the software. Given that programming accounts for only a small percentage of the actual cost of large software developments - does this matter? The answer is yes for two reasons:

First, although it is minor part of the total effort it is the principle raison d'etre for

the whole process. This means that an improvement in programming productivity leads to corresponding reductions in all the other dependent activities.

Secondly, cost reduction is not the only important issue to the customer. An equally important aspect is the fitness for purpose and the reliable operation of the software system. Techniques such as formal specification of software systems and fast prototyping techniques are being advocated as a means of tackling these problems. Current developments in PTI technology provide little or no support for these techniques which are intimately connected with programming and implementation methods, and for which the performance overhead of using the high-level entity/relationship/attribute model could pose a serious problem. It is exactly these areas which PTIs do not address which are covered by the Ten15 kernel.

4 CURRENT STATUS AND FUTURE PLANS

In order to allow the properties of the Ten15 high-integrity kernel to be assessed, RSRE are working on a Ten15 evaluation system. This is an extensible, advanced programming environment comprising:

- A simple but powerful Human/Computer Interface which will be based on the experience gained with the editor from the RSRE FLEX PSE [6]. This will use an advanced hypertext format and will be user-extensible.
- Compilers for Pascal. Algol68 and Ten15-notation. Ten15-notation is the main implementation language of the evaluation system and serves as both an assembler for Ten15 (in that it allows text to be written that will generate exactly any piece of Ten15 required) and as a high-level system programming language.
- A symbolic Ten15 debugger which the compilers for the different languages tailor to their individual syntax.
- A separate compilation system.
- A framework of tools that allows the algebraic structure of Ten15 programs to be manipulated by user-written programs.

The initial release of this system will be as a standalone environment on VAX/VMS or SUN3. Later versions will extend the facilities to heterogeneous networks of Ten15 machines.

The evaluation system is already in use within RSRE for evaluating Ten15's potential for support of secure systems. The evaluation system should be ready for use with collaborators outside RSRE towards the end of 1991. We hope to attract interest from research teams in academia, industry, or other governmental research establishments, who could use Ten15 to provide significant extra leverage to design challenging applications on top of Ten15.

References

[1] Dr J M Foster. The Algebraic Specification of a Target Machine: Ten15, High-integrity software. ed C T Sennett, Pitman. 1989.

[2] S R Wiseman. H S Field-Richards. The SMITE Computer Architecture, RSRE Memo No 4125. Jan 1988.

[3] Dr W J Cullver. Dr C H Pygott. Application of Formal Methods to the VIPER Microprocessor. IEE Proc Vol 134. pt E. No 3. May 1987.

[4] S R Wiseman. Basic Mechanisms for Computer Security. RSRE Report No 89024, Jan 1990.

[5] Introducing PCTE+. Independent European Programme Group, Technical Area 13, 1989.

[6] Mrs M Stanley. An Evaluation of the FLEX Programming Support Environment, RSRE Report 86003. Aug 1986.

NATO UNCLASSIFIED

REPORT DOCUMENTATION PAGE

1. Recipient's Reference:	2. Further Reference:
3. Originator's Reference: AC/243(Panel 11)TP/1	4. Security Classification: UNCLASSIFIED/UNLIMITED 5. Date: 15.04.91 — 6. Total Pages: 10

7. Title (NU):

Formal Program Developments

8. Presented at:

AC/243(Panel 11) Symposium on Military Information Systems Engineering - RSRE, Malvern, UK - 8-10 May 1990

9. Author's/Editor's:
J. Cazin, R. Jacquart, M. Lemoine and P. Michel

10. Author(s)/Editor(s) Address:
ONERA-CERT/DERI
2, Avenue E. Belin
B.P. 4025
31055 Toulouse CEDEX
France

11. NATO Staff Point of Contact:
Defence Research Section
NATO Headquarters
B-1110 Brussels
Belgium
(Not a Distribution Centre)

13. Keywords/Descriptors:

FORMAL DEVELOPMENTS, DEVELOPMENT LANGUAGE, TYPE SYSTEMS, TYPED LAMBDACALCULUS, DEVA, REUSE OF DEVELOPMENTS

14. Abstract:

Formalizing program development aims at mastering the correctness of programs, controlling the application of methods and tackling reusability issues.

The development language DEVA is introduced, and the type system it is based on is overviewed. Examples of use of DEVA to formally express some developments are given. Reuse aspects related to formal developments are illustrated.

FORMAL PROGRAM DEVELOPMENTS

J.Cazin*, R. Jacquart*, M. Lemoine*, P.Michel*

CONTENTS

* ONERA-CERT/DERI
2, Avenue Edouard Belin, 31055 Toulouse Cedex - FRANCE
Tel: (+33) 61 55 70 55
Fax: (+33) 61 55 71 12
Email: cazin@tls-cs.cert.fr

1. INTRODUCTION

A few years ago, formal developments were still considered an academic task. The main reasons were the size of addressable problems, the heaviness of notations, the lack of support tools, and the time overhead compared with usual empiric developments. These few arguments become more and more obsolete excepted for the last one: formalizing developments is still ressource over consuming. Nevertheless we can observe a growing interest of industrial companies for formal methods applied to real scale problems. Real experiences like compiler construction, user interface development of safety critical systems, security systems development are reported in [8]. Several reasons can explain this change of interest:

1. formalizing developments aims at *mastering the correctness* of the developed objects. This is specially the case for safety critical systems. It has been observed that usual systematic testing methods become less and less applicable when the complexity of systems grows. Indeed, tests themselves must be formally specified to be credible, and this activity is also time consuming. In this case formal developments become competitive.
2. it is in common agreement that *rigorous methods* are mandatory to be used to produce correct software. But it makes no use to define such methods without giving means to *control their application*. Such a control must dispose of a formal notion of development it can refer to when checking the correctness of a given development step.
3. developments, when totally formalized, give a complete precise description of the set of steps leading from a specification to the corresponding program. If they are represented by terms which can be manipulated by higher order functions, they become *reusable* and applicable to different close problems. Then the overcost becomes acceptable.

The present paper is based on some partial results of two European projects: ToolUse and REPLAY[2]. The first one – **ToolUse** [2]– aimed at producing a software development environment offering a high level of parameterization. In this environment, software development methods are formally defined, and their application is checked. This implies the formalization of the developments themselves. A major issue of the project is the development language **DEVA** which will be overviewed in section 2. Examples of use of this language to formally express developments are given in section 3.

The second project – **REPLAY** [3] – aims at studying reusability of formalized *developments*. This is an original point of view compared to the usual practice of reusing developed *components*. The approach will be shortly illustrated in sub-section 3.5

2. THE DEVELOPMENT LANGUAGE DEVA

2.1. Technical basis

A major characteristics of the environment developed in the ToolUse project is to be *institution free*. That means that notations and tools which are the basis of the environment

[2] ToolUse and REPLAY are partially funded by ESPRIT-I programme

are not devoted to a specific method or to particular specification and programming languages. As a consequence, the DEVA language has been defined as a very general notation to support a calculus on developments.

DEVA can be briefly described as a typed lambda-calculus with dependent types. The main works which influenced its definition have been the Automath project [6], the Calculus of Constructions [4], and Intuitionistic Type Theory[5]. The general idea driving the definition is to give means to describe the developed objects, the development steps and the underlying rules as terms (the so-called *texts* introduced in section 2.2) and to use a general typing mechanism to control the correctness of these objects. Moreover developments, and underlying theories can be structured and organized using the *context* expression introduced in section 2.3.

The basic elements supported by DEVA are briefly introduced in the following sections, together with the main rules of the typing mechanism. A complete description of the language, and a formal definition of its semantics can be found in [7].

In the following we shall use the following syntactic conventions:

x,y,z,xi,yi,zi ... stand for variable symbols,

t,ti,tt,tti ... are DEVA texts. More often *tti* is used to denote the type of the text *ti*,

c,ci ... are any DEVA context.

2.2. DEVA *texts*

The texts objects are the basic entities supported by DEVA. Each text is typed and types are themselves typable texts. So we dispose of a recursive system supporting dependent types, which starts with an initial text *primal* which is untypable.

construction	syntax	comments
initial text	*primal*	the only untypable DEVA text
symbol	*x*	x must be declared or defined (see contexts)
abstraction	*[c/-t]*	the usual lambda-abstraction
application	*t1(t2)*	t1 must be an abstraction over the type of t2
judgement	*t1.:t2*	the type of t1 is judged to be t2
sum	*[t1/t2/.../tn]*	
product	*[x1:=t1, ... ,xn:=tn]*	xi can be used as projection operators

Table 1 Main *text* constructors

Some features are supported by DEVA to express control. They are also expressed by means of texts which can be manipulated as other ones with the same rules.

construction	syntax	comments
sequential composition	*t1 o t2*	the usual functional composition
case distinction	*case t1 of t2*	t1 is a sum and t2 a corresponding product
iteration	*loop t*	stops when the result is applicable

Table 2 constructors for *control texts*

2.3. DEVA *contexts*

The notion of context allows to structure the developments expressed in DEVA. Contexts are used to express theories the developments are based on: for example, basic mathematical theories, logics, algebraic data-types, specification and programming languages, and rules constituting a method.

Contexts are built up as sequences of declarations allowing to introduce new typed symbols. Definitions allow to give a name to a given text, and can be considered as abbreviations. Moreover, contexts can use each other through the mechanism of importation which gives access to the declarations, and definitions present in another context.

construction	syntax	result
empty context	*nilc*	
text declaration	*x:t*	x is a new symbol of type t
text definition	*x:=t*	x is a new symbol abbreviating the text t
implicit definition	*x?t*	x is a text of type t to be synthesized
sequential composition	*[:c1;c2/]*	the two contexts c1 and c2 are appended
context definition	*part p := c*	p is a new symbol abbreviating the context c
context importation	*import c*	the symbols visible in c become visible
context application	*c(t)*	c where the first declaration is substituted by t
symbol renaming	*c[x=:y]*	c where x is renamed to y
symbol hiding	*c\x*	c where x is no more usable

Table 3 main *context* constructors

The scoping rules are fairly simple: each symbol introduced at a given place in a sequence is usable in the following of the sequence, symbols defined in a given context are usable after the importation of that context, importation is transitive.

2.4. Typing rules

DEVA texts can be considered as λ–terms of a typed λ–calculus. A text is *well-formed* if it is built up using correct construction of texts introduced hereabove. A *well-formed* text is valid if it is *well-typed*. A context is valid if it is made of declaration and definitions of valid texts.

The following rules aim at giving some intuition of the typing mechanism supported by DEVA. $V_c\{c_1, c_2\}$ states the relative validity of a context c_2 in the context[3] of c_1. $V_a\{c\}$ represents the absolute validity of a context outside any other context. $V_t\{c, t\}$ is the validity of a text t in a given context c. $E_t\{c, t_1, t_2\}$ is the equivalence of two texts t_1 and t_2 in a given context c and $T(c, t)$ the type of the text t in the context c.

Some rules stating *validity of texts* can be given to illustrate typing mechanism:

$$R_{t1} : V_t\{c, \text{primal}\}$$

[3] In the following, "in the context of c" must be understood as "in the situation where the components of the context c are visible"

$$R_{t2}: \frac{V_c\{c_1, c_2\} \quad V_t\{[|c_1; c_2|], t\}}{V_t\{c_1, [c_2 \vdash t]\}}$$

$$R_{t3}: \frac{V_t\{c, [x : tt_1 \vdash t_2]\} \quad V_t\{c, t_1\} \quad E_t\{c, T(c, t_1), tt_1\}}{V_t\{c, [x : tt_1 \vdash t_2](t_1)\}}$$

$$R_{t4}: \frac{V_t\{c, t_1\} \quad V_t\{c, t_2\} \quad E_t\{c, T(c, t1), t_2\}}{V_t\{c, t_1 \therefore t_2\}}$$

...

R_{t1} rule introduces validity of the initial text in any context. R_{t2} rule states the validity of an abstraction provided that the abstraction context c_2 is valid in the context c_1 and that the abstracted text is valid in the sequential composition of c_1 and c_2. R_{t3} rule states the validity of a text application: each argument must be valid in the current context c, and the applied text $[x1 : tt_1 \vdash t_2]$ must be an abstraction the first element of which is a declaration of a type tt_1 equivalent to the type of the text t_1 on which the application is done. R_{t4} rule states the validity of a judgement: the type of the first argument must be equivalent to the second one.

Validity of contexts can be described in a similar manner:

$$R_{c1}: V_a\{\text{nilc}\}$$

$$R_{c2}: V_a\{c\} \equiv V_c\{\text{nilc}, c\}$$

$$R_{c3}: \frac{V_a\{c\}}{V_c\{c, \text{nilc}\}} \qquad R_{c4}: \frac{V_a\{c\} \quad V_c\{c, c_1\} \quad V_c\{[|c; c_1|], c_2\}}{V_c\{c, [|c_1; c_2|]\}}$$

$$R_{c5}: \frac{V_a\{c\} \quad V_t\{c, t\}}{V_c\{c, x : t\}} \qquad R_{c6}: \frac{V_a\{c\} \quad V_t\{c, t\}}{V_c\{c, x := t\}} \qquad R_{c7}: \frac{V_a\{c\} \quad V_t\{c, t\}}{V_c\{c, x?t\}}$$

$$\vdots$$

The absolute validity is true for the empty context (R_{c1} rule) and it amounts to relative validity in empty context for any other context c (R_{c2} rule). R_{c3} and R_{c4} rules allow to decide the validity of a context by checking the validity of its sub-contexts. R_{c5}, R_{c6} and R_{c7} rules allow to introduce new declarations or definitions of text t, provided that this text is valid in the current context c.

In order to make following sections understandable, the definition of the typing function must be also sketchy introduced for symbols, abstraction and application:

$$T(c, \text{primal}) \equiv \text{undef} \qquad T(\text{nilc}, x) \equiv \text{undef}$$

$$T([|c; y : t|], x) \equiv \begin{cases} t \text{ if } x \text{ and } y \text{ are the same symbol} \\ T(c, x) \text{ otherwise} \end{cases}$$

$$T([|c; y := t|], x) \equiv \begin{cases} T(c, t) \text{ if } x \text{ and } y \text{ are the same symbol} \\ T(c, x) \text{ otherwise} \end{cases}$$

$$T \equiv [c_1 \vdash T([|c; c_1|], t)] \qquad T(c, t_1(t_2)) \equiv T(c, t_1)(t_2)$$

...

Finally the equivalence of 2 texts $E_t\{c, t_1, t_2\}$ is defined as the transitive closure of reduction operations. These rules are a generalisation of the usual β-reduction which is the basis of the λ-calculus. It is not detailed here.

3. FORMAL EXPRESSION OF DEVELOPMENTS WITH DEVA

3.1. Formal developments in a transformational approach

The developments that will be manipulated in the following are based on a transformational approach. In such a framework, the developed objects constitute a continuum from specification to program, and one object is produced from previously existing ones by applying elementary transformations.

The transformations we shall use are based on the *fold/unfold* system, originally developed by Burstall and Darlington [1]. It allows to transform programs specified as a set of equations down to the level of programming language, and it is specially tailored for developing programs in an applicative style.

To develop a program in such a framework, we have to dispose of an initial rigorous specification. In our examples, this will be formulated in classical set theory. We do not worry about producing this early specification which is a problem of requirements engineering, out of the scope of our work. The development will be expressed in a context summarized on figure 1 in which:

- each data-type involved in the development is typed. Types are defined in a specific DEVA context *Sorts*
- the *Equations* context defines properties of equations betwen objects having the same type
- elementary data-types like *Sets, Pairs, propositions* are defined in specific contexts
- *Specific Functions* contains the definitions of functions which are in common use although not defined in basic data-types (like *Filter, Map*, aso...)

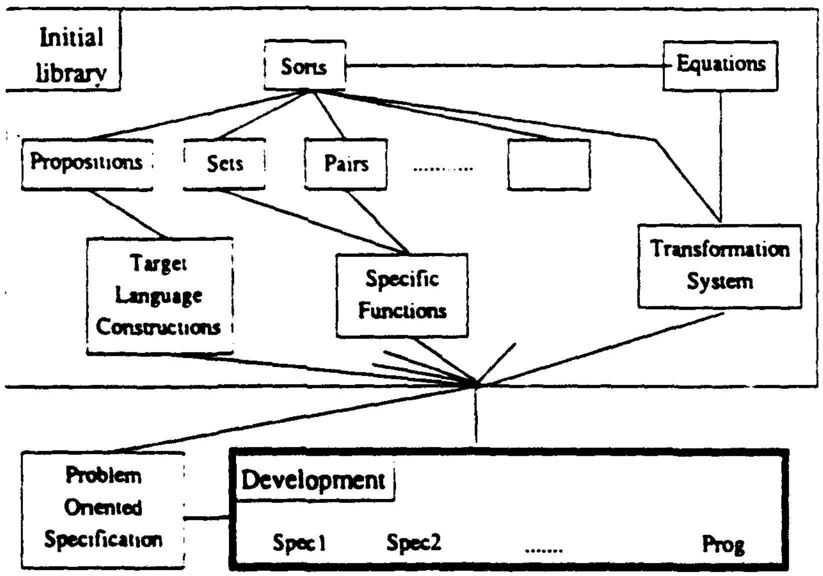


Figure 1 A schematic development and its context

- *Target Language Constructions* allows to introduce in a progressive manner, the basic expressions the developed program will be made of.

3.2. An example of a problem oriented specification

The example given hereafter is a rigourous specification of the solution of a real life problem (typing of human cells and serum in biological area).

$$\mathcal{E}^{+} \triangleq \{p \in \mathcal{P}(\mathcal{A}) \mid (p,+) \in \mathcal{R}\}$$

$$\mathcal{E}^{-} \triangleq \{m \in \mathcal{P}(\mathcal{A}) \mid (m,-) \in \mathcal{R}\}$$

$$H \triangleq \left\{p \in \mathcal{E}^{+} \mid \forall x \in p . \exists m \in \mathcal{E}^{-} . x \in m\right\}$$

$$Cred \triangleq \{a \in \mathcal{A} \mid Cred(a,\mathcal{R})\}$$

$$\mathcal{E}'^{+} \triangleq \left\{p \in \mathcal{E}^{+} \mid p \cap \mathcal{A}_{cred} = \emptyset \Rightarrow \neg(p \in H)\right\}$$

$$\mathcal{E}'^{-} \triangleq \left\{m \in \mathcal{E}^{-} \mid \forall p \in H . m \cap p \cap \mathcal{A}_{cred} = \emptyset\right\}$$

$$H' \triangleq \left\{p \in \mathcal{E}'^{+} \mid \forall x \in p . \exists m \in \mathcal{E}'^{-} . x \in m\right\}$$

$$\mathcal{E}''^{+} \triangleq \mathcal{E}'^{+} - H'$$

$$Result \triangleq \left\{ q \in \mathcal{P}(\mathcal{A}) \mid \exists p \in \mathcal{E}''^{+} . q = p - \bigcup_{m \in \mathcal{E}'^{-}} m \right\}$$

Although this expression is short, the problem is totally specified and the corresponding program has been formally developed in the REPLAY project so as to give a basis for reusability experiments. In the following, we will only focus on the two first expressions.

3.3. A piece of development

The part of the development corresponding to the computation of the first set is given in a semi-formal understandable way:

$$\begin{array}{ll}
\mathcal{E}^{+} \triangleq \{p \in \mathcal{P}(\mathcal{A}) \mid (p,+) \in \mathcal{R}\} & \\
\hline
\mathcal{E}^{+} \triangleq \{p\ s_A \mid (p,+) \in \mathcal{R}\} & Int_{set} \\
\hline
\mathcal{E}^{+} \triangleq \{p\ s_A \mid \exists y\ r s_{A\Pi} . y \in \mathcal{R} \wedge y = (p,+)\} & \exists_{intro} \\
\hline
\mathcal{E}^{+} \triangleq \{p\ s_A \mid \exists y\ r s_{A\Pi} . y \in \mathcal{R} \wedge (p = \Pi_1(y) \wedge + = \Pi_2(y))\} & \Pi_{intro} \\
\hline
\mathcal{E}^{+} \triangleq \{p\ s_A \mid \exists y\ r s_{A\Pi} . y \in \mathcal{R} \wedge (+ = \Pi_2(y) \wedge p = \Pi_1(y))\} & \wedge_{comm} \\
\hline
\mathcal{E}^{+} \triangleq \{p\ s_A \mid \exists y\ r s_{A\Pi} . (y \in \mathcal{R} \wedge + = \Pi_2(y)) \wedge p = \Pi_1(y)\} & \wedge_{assoc} \\
\hline
\mathcal{E}^{+} \triangleq \{p\ s_A \mid \exists y\ r s_{A\Pi} . y \in Filter(\lambda x . \Pi_2(x) = +, \mathcal{R}) \wedge p = \Pi_1(y)\} & Filter_{intro} \\
\hline
\mathcal{E}^{+} \triangleq Map(\Pi_1, Filter(\lambda x . \Pi_2(x) = +, \mathcal{R})) & Map_{simpl}
\end{array}$$

where $\frac{Exp_{in}}{Exp_{out}} Rule$ must be understood as applying *Rule* (with right substitutions) to a sub-expression of Exp_{in} produces Exp_{out}.

3.4. Expression using DEVA

To formally express this development with DEVA, we must first give an expression for the elementary operation (the *"unfold"*) allowing to replace the left hand side of an equation by its right hand side inside the right hand side of another equation.

$$unfold: \left[\mid \frac{s_1, s_2 ?sorts;\ x_1 ? s_1;\ x_2, y_2 : s_2;\ f ? [s_1 \vdash s_2]}{\mid \frac{x_2 = y_2; \qquad x_1 = f(x_2)}{x_1 = f(y_2)}} \right]$$

To use the first rule, we have only to apply it to two arguments whose the DEVA types are $x_2 = y_2$, and $x_1 = f(x_2)$. The other parameters will be synthesized, and we shall get something of type $x_1 = f(y_2)$.

Let us consider the first step of development

$$\frac{E^+ \triangleq \{p \in \mathcal{P}(\mathcal{A}) | (p,+) \in \mathcal{R}\}}{E^+ \triangleq \{p\ s_A | (p,+) \in \mathcal{R}\}} Int_{set}$$

It expresses that starting from a specification of E^+, we decide to keep the information $p \in \mathcal{P}(\mathcal{A})$ only as a typing information (i.e. we represent the powerset $\mathcal{P}(\mathcal{A})$ in intention). So we must represent with DEVA:

1. the first level of sets $\{x \in S | P(x)\}$ this will be done with the constructor:

   $$S_0 : [s ? sort;\ S : set(s);\ P : [s \vdash prop] \vdash set(s)]$$

2. the second level where we decide of a representation in extension or in intention, and where we keep typing information: $\{x\ _s | P(x)\}$ will b represented with another constructor:

   $$S_1 : [s ? sort;\ P : [s \vdash prop] \vdash set(s)]$$

3. the rule representing the design decision to represent a set in intention:

   $$intset : \left[\frac{s ? sort;\ S : set(s);\ P : [s \vdash prop]}{S_0(S, P) = S_1(P)} \right]$$

4. the initial definition of E^+ with the corresponding type:

   $$E^+ : set(e)$$

   $$def E^+ := E^+ = S_0(pow(\mathcal{A}), [p : set(e) \vdash isin((p,+), \mathcal{R})])$$

5. the first step consist then in unfolding the whole right hand side of the equation using the $intset$ rule:

   $$step_1 := unfold(intset(pow(\mathcal{A}), [p : set(e) \vdash isin((p,+), \mathcal{R})]), def E^+)$$

   $$\therefore\ E^+ = S_1([p : set(e) \vdash isin((p,+), \mathcal{R})])$$

   and so, we get the result as the type of the resulting application (judgement has been used to exhibit this type).

The whole development can be expressed the same way, each elementary step build a *text* the type of which is an equation which will be itself the type of input argument of the next step.

$$; step_1 := unfold(intset(\ldots), defE^+) \therefore E^+ = S_1([p:set(a) \vdash isin((p,+),\mathcal{R})])$$

$$; step_2 := unfold(exintro(\ldots), step_1)$$

$$\therefore E^+ = S_1([p:set(a) \vdash exist([y:pair(set(a),res) \vdash (isin(y,\mathcal{R}) and (y=(p,+)))])])$$

$$; step_3 := unfold(piintro(\ldots), step_2)$$

$$\therefore E^+ = S_1(\ldots)$$

$$\vdots$$

$$; step_7 := unfold(mapsimpl(\ldots), step_6)$$

$$\therefore E^+ = map(pi1, filter([x:pair(set(a),res) \vdash pi_2(x)=+], \mathcal{R}))$$

3.5. Reuse of formal developments

An interesting application of formally described developments is to reuse them to deal with problems close to each other. In this case the formal expression of the early development must be modified to take into account characteristics of the new problem. This is done in DEVA using higher order function. For example, in the problem oriented specification introduced in sub-section 3.2, we can notice that the specifications of E^- can be deduced from the specification of E^+ only by changing + to –, and the corresponding definition $defE^+$ to $defE^-$. This can be simply expressed in DEVA usin meta abstractions followed by instanciation on the right values:

$$; E^- : set(a)$$

$$; defE^- := E^- = S_1(powA), [m:set(a) \vdash isin((m,-),\mathcal{R})])$$

$$; reuseddev := ABS(ABS(step_7, defE^+), +)(-, defE^-)$$

$$\therefore E^- = map(pi1, filter([x:pair(set(a),res) \vdash pi_2(x) = -], \mathcal{R}))$$

So we get in one step a correct development for the second set.

This operation is fully general and can be applied on substantial pieces of development, to discard some particular parts and replace them by other.

4. CONCLUSION

The DEVA language overviewed in this paper is a powerful notation which allow to express the semantics of development in a totally formal way. The examples given in this paper are very sketchy, but we can mention that DEVA is able to support significant examples like some description of VDM and JSP/JSD methods.

The DEVA language is supported by an evaluator which can be considered as a high level type-checker. It forbids the user to cheat with the application of a method, enforcing him to formally describe all the developments steps he follows.

What may look like a heavy task in a first shot development is, on the contrary, fruitful in the context of reuse, and allows to get in a few steps programs correct wrt their specifications.

Up to now, DEVA has been mainly used to address the problem of developments of correct sequential programs. In the future, it will be used to support formal developments of concurrent embedded systems.

References

[1] R.M. Burstall and J.Darlington. A transformation system for developing recursive programs. *JACM*, 24(1):44–67, 1977.

[2] J. Cazin, R. Jacquart, M. Lemoine, P. Maurice, and P. Michel. Method driven programming. In *IFIP 89*, 89.

[3] J. Cazin, R. Jacquart, M. Lemoine, and P. Michel. Manipulation of formal developments expressed in deva. In K.H. Bennett, editor, *Software Engineering Environments*. Ellis Horwood, 89.

[4] T. Coquand and G. Huet. Constructions: a higher order proof system for mechanizing mathematics. In *EUROCAL 85*, 1985.

[5] P. Martin-Lof. Constructive mathematics and computer programming. In Hoare and Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 167–184. Prentice Hall, 85.

[6] R.P. Nederpelt. *An Approach to Theorem Proving on the Basis of a Typed Lambda Calculus*. Springer Verlag, LNCS 87, 1980.

[7] M. Sintzoff, M. Weber, Ph. de Groote, and J. Cazin. Definition 1.1 of the generic development language deva. Technical report, Esprit, 89.

[8] J.C.P. Woodcock. Formal techniques and operational specifications. *SEE notes*, 14(5), 89.

REPORT DOCUMENTATION PAGE

1. Recipient's Reference:	2. Further Reference:
3. Originator's Reference: AC/243(Panel 11)TP/1	4. Security Classification: UNCLASSIFIED/UNLIMITED 5. Date: 15.04.91 — 6. Total Pages: 11

7. Title (NU):

Automated Support for Development and Evolution of Complex Software Systems

8. Presented at:

AC/243(Panel 11) Symposium on Military Information Systems Engineering - RSRE, Malvern, UK - 8-10 May 1990

9. Author's/Editor's:
Jack C. Wileden

10. Author(s)/Editor(s) Address:
Dept. of Computer and Information Science
Lederle Graduate Research Centre
University of Massachusetts
Amherst, MA 01003, USA

11. NATO Staff Point of Contact:
Defence Research Section
NATO Headquarters
B-1110 Brussels
Belgium
(Not a Distribution Centre)

12. Distribution Statement:

13. Keywords/Descriptors:

ENVIRONMENT, OBJECT MANAGEMENT, TYPE MODELS, PERSISTENCE, INTEROPERABILITY, CONCURRENCY ANALYSIS, CONSTRAINED EXPRESSIONS, AUTOMATED TOOLSET

14. Abstract:

The demand for ever greater levels of reliability in ever more complex software systems, especially those that are highly concurrent, distributed, or subject to stringent real-time constraints, demands increasingly powerful automated support for software developers. For the last several years, our work, in collaboration with our colleagues in the Arcadia consortium, has been directed toward the development of advanced software environment and tool technology that will provide the necessary automated support for software developers. Two major foci of our work have been (1) the object management capabilities needed in a basis integrating infrastructure for advanced software environments and (2) analysis tools applicable to concurrent, distributed and real-time software.

In this paper, we summarize our work in these areas, indicate how it fits into the larger Arcadia project framework, and suggest future directions for efforts aimed at developing advanced software environment and tool technology.

AUTOMATED SUPPORT FOR DEVELOPMENT AND EVOLUTION OF COMPLEX SOFTWARE SYSTEMS

Jack C. Wileden*

Contents

* Associate Professor of Computer and Information Science,
University of Massachusetts, Amherst, Massachusetts 01003 USA

1 INTRODUCTION

The demand for ever greater levels of reliability in ever more complex software systems, especially those that are highly concurrent, distributed, or subject to stringent real-time constraints, demands increasingly powerful automated support for software developers. For the last several years, our work, in collaboration with our colleagues in the Arcadia consortium, has been directed toward the development of advanced software environment and tool technology that will provide the necessary automated support for software developers. Two major foci of our work have been 1) the object management capabilities needed in a basic integrating infrastructure for advanced software environments and 2) analysis tools applicable to concurrent, distributed and real-time software.

In the area of object management capabilities for environments, we have developed a variety of type definition models and mechanisms. These have been specifically tailored for use in describing the components of software environments and also the components of software systems that could be developed using those environments. We have also developed and implemented an approach to adding persistence as an orthogonal property of typed objects. We believe that this approach to persistence provides the basis for moving from environments based on files to environments organized around a space of persistent typed objects. Finally we have developed the Specification Level Interoperability approach to supporting interoperation of software written in several different languages. We have produced an initial prototype implementation of this approach to interoperability that demonstrates its effectiveness by integrating a collection of software development tools written in Ada and Lisp.

In the area of analysis tools applicable to concurrent, distributed and real-time software, we have developed, and built a prototype toolset supporting, the Constrained Expression approach. Constrained expressions are a language-independent, event-based, closed-form representation for behavior of systems. We believe that this representation is particularly appropriate for describing concurrent, distributed or real-time system behaviors. Our current prototype toolset supports the analysis of concurrent systems described in an Ada-like design language. The results of our initial experiments with the toolset have been very encouraging. These results indicate that the constrained expression approach has the potential to be of practical use in analyzing realistic problems in concurrent or distributed software design. We have also carried out an initial experiment in analyzing some timing properties of a concurrent system. Our success in this experiment has led us to continue investigating the applicability of the constrained expression approach and tools to real-time system problems.

In the remainder of this paper, we give a brief overview of the Arcadia project, summarize our own work in the areas of object management and analysis tools for concurrent software, and suggest future directions for efforts aimed at developing advanced software environment and tool technology.

2 ARCADIA AND SDL

The Arcadia project [15] is a collaborative software environment research program encompassing groups at several universities and industrial organizations, including the Software Development Laboratory (SDL) at the University of Massachusetts. The objective of Arcadia is to develop advanced software environment technology and to demonstrate this technology

through prototype environments. The initial Arcadia environment prototypes are being built primarily in Ada and targetted primarily to support Ada software development.

The principal research areas being addressed within the Arcadia project are environment architecture, user interface management, support for process definition and actualization, object management, support for measurement and evaluation, analysis techniques and tools, and language processing tools. Various researchers at various of the Arcadia sites are involved in working on each of these areas. In the present paper, we summarize only a subset of the work on object management and analysis tools that has been carried out within SDL. Readers interested in a more comprehensive overview of Arcadia research in these two areas or in other areas of Arcadia research are referred to [15] and the papers cited therein.

3 OBJECT MANAGEMENT

Extensibility, integration and interoperability are three important goals for Arcadia environments. Much of our recent effort in SDL has been directed toward the definition and implementation of support for the object management capabilities that we believe to be crucial for building integrated and extensible environments supporting tool interoperability. One focus of this research has been type models for (persistent) object management in environments. Another primary focus of the research has been on implementing persistent objects in the context of strong abstract typing. Our final focus has been on approaches to supporting tool interoperability, environment extensibility and integration. We briefly describe these efforts in the following subsections.

3.1 Type Models

One component of our work on typing support for environment developers has been the definition of a type model that we call Oros [13] (We distinguish between *type system*, a specific collection of types developed for use in some application (such as a particular software environment or tool), and *type model*, a framework or mechanism for defining type systems.) The Oros type model is intended to permit environment builders and users to define types for environment components, such as tools or process programs, as well as types for software product components, such as requirements, designs, code, plans, and so forth. We believe that such pervasive typing can play a central role in improving the organization, increasing the reliability and facilitating the evolution of environments and the software systems that they are used to build.

The Oros type model supports both the definition of types and the determination of inter-type relationships. Our choices of the primitive types, type definition scheme and type constructors for Oros reflect our views concerning the fundamental kinds of entities that make up a software environment, the equally important roles of *relationships* and *operations* in defining the types of those entities, and the need for precise, powerful and flexible specification of inter-type relationships. We are currently experimenting with the application of these facets of the Oros model to the description of various environment components to assess the accuracy of our views and the efficacy of the model. We are also designing prototype implementations of Oros. Finally, as described in section 3.3, we are employing a subset of Oros in a prototype implementation of our approach to supporting interoperability. Through experimenting with

such prototypes and refining OROS we expect to produce a specification of the type modelling capabilities needed in advanced software development environments such as Arcadia.

3.2 Persistence

We believe that the availability of a persistent object store, smoothly integrated into the language(s) used by environment and tool builders, will dramatically simplify the building of environments. Our research on persistence has been directed toward 1) identifying an appropriate set of abstractions through which environment designers and tool builders can manipulate persistent objects, and 2) exploring implementation strategies for persistence.

The PGRAPHITE system [17] is our currently operational prototype of a persistent object capability This system is a preprocessor that accepts definitions of abstract graph data types, specified in the Graph Definition Language (GDL) [7], and produces an Ada implementation of the specified graph types incorporating orthogonal persistence. PGRAPHITE provides environment designers and tool builders with three sets of abstractions for manipulating persistent objects, namely *persistent object*, *persistent store* and *graph* abstractions. Our persistent object abstraction augments the operations available on any type with operations that can make individual objects of that type become persistent. Hence the persistence property is orthogonal to any other properties of a type, persistence can be controlled on an instance by instance basis, and tools interact with a persistent store containing persistently typed objects, rather than with a file system containing only one persistent type (namely file). Under the PGRAPHITE persistent store abstraction, tools can access persistent stores, which are called repositories, only during a *session*, and a tool must explicitly indicate the beginning and end of sessions. Sessions provide a basis for concurrency management and hence support sharing of persistent graph nodes, both among tools and between two or more graphs.

Our implementation strategy has several interesting features. In order to preserve abstract typing and information hiding, object classes manage their own persistence in PGRAPHITE. Efficiency of both memory utilization and I/O traffic is increased through fault-driven retrieval of objects. The PGRAPHITE processor automatically generates Ada implementations of abstract types and repository managers. While this automatic generation capability is currently available only for abstract graph types describable in GDL, the approach is completely general and we have manually applied it to a wide range of types. We have also ported our PGRAPHITE implementation to several different underlying storage managers, including Ada Direct_IO and the Mneme system [12]. This porting is facilitated by the standardized Storage Manager Interface that is part of the PGRAPHITE implementation architecture.

Through use of the PGRAPHITE system, both within SDL and at several other sites, we are exploring several important issues concerning persistent typed object management in environments. We are currently working to extend PGRAPHITE by automating the generation of additional types, and by adding support for concurrency, version control and garbage collection. We are also investigating "principled" approaches to limiting the extent of persistence, which will complement the reachability-based definition of extent of persistence that is provided by PGRAPHITE [19].

3.3 Interoperability

There is an increasing need and desire to develop systems, and especially environments, by combining components that are written in different languages and/or that are run on different kinds of machines. Success at this depends in large part on the *interoperability* of the components—that is, the ability of the components to communicate and work together despite their differing backgrounds. While most previous approaches to interoperability have provided support at the representation level, we are pursuing an approach that will provide support at the specification level. We have developed a model of Specification Level Interoperability (SLI) [18] that consists of four components: 1) a *unified type model*, which is a notation for describing the entities to be shared by interoperating programs; 2) *language bindings*, which connect the type models of the languages to the unified type model; 3) *underlying representations and implementations*, which realize the types used by the different interoperating programs; and 4) *automated assistance*, which generally eases the task of combining components into an interoperable whole. To demonstrate and investigate SLI, we have created a prototype realization of the approach and applied it to achieving interoperability of several components of the constrained expression toolset described in section 4.2.

The unified type model (UTM) concept is central to the SLI approach. The intent is that a UTM should serve as a basis for tool cooperation within a richly structured collection of environment components. Integration will be enhanced by permitting the tools to share and exploit the rich structure of those components. This is in direct contrast to the Unix model, for example, a representation level approach to interoperability that forces tools to interact at the level of byte streams. The richer structure supported by a UTM will permit consistency checking (e.g., type checking) and will free tools from the necessity of explicitly translating (parsing and/or unparsing) inputs and outputs. A UTM can be viewed as a semi-strong coupling of environment components - stronger than typeless byte streams, but weaker than a single fixed type system. We believe that this intermediate position is the key to simultaneously attaining interoperability, integration and extensibility.

The UTM included in our initial prototype realization of SLI represents an attempt to be compatible with a variety of existing or proposed type models and type systems and is also intended to be appropriate for short term practical use in Arcadia prototypes. This initial UTM is based upon a simple subset of OROS concepts. It consists of a set of type definition primitives, a set of relationships or constructor functions, a set of "special" types, and some semantics for manipulation of instances. We have successfully used this UTM in our prototype SLI realization to describe and support automatic generation of multilingual implementations of an object type that is central to the constrained expression toolset.

4 ANALYSIS TOOLS FOR CONCURRENT SOFTWARE

A wide variety of techniques have been proposed for analyzing the behavior of concurrent software systems. These differ in their underlying models of concurrent computation, in the questions about behavior they attempt to answer, and in the stages of the software development process in which they are applied. It is, *of course*, unlikely that any single approach to analysis can possibly meet all the needs of software developers throughout the development process. Therefore, the goals of the Arcadia project include developing, and facilitating the integration

of, various approaches to analyzing the behavior of concurrent software systems.

In the following subsections we briefly describe one such approach, called the Constrained Expression approach, a prototype toolset supporting that approach, and the results of some preliminary experiments with the toolset.

4.1 Constrained Expressions

In the constrained expression approach to analysis of concurrent systems, the system descriptions produced during software development (e.g., designs in some design notation) are translated into formal representations, called *constrained expression representations*, to which a variety of analysis methods are then applied. This approach allows developers to work in the design notations and implementation languages most appropriate to their tasks. Rigorous analysis is based on the constrained expression representations that are mechanically generated from the system descriptions created by software developers.

This subsection contains a brief overview of the constrained expression formalism. A detailed and rigorous presentation is given in [8], and a less formal treatment presenting the motivation for many of the features of the formalism appears in [4]. The use of constrained expressions with a variety of development notations is illustrated in [4] and [10].

The constrained expression formalism treats the behaviors of a concurrent system as sequences of events. These events can be of arbitrary complexity, depending on the system characteristics of interest and the level of system description under consideration. Associating an *event symbol* to each event, we can regard each possible behavior of the system as a string over the alphabet of event symbols.

We use interleaving to represent concurrency. Thus, a string representing a possible behavior of a system that consists of several concurrently executing components is obtained by interleaving strings representing the behaviors of the components. The events themselves are assumed to be atomic and indivisible. "Events" that are to be explicitly regarded as overlapping in time are represented by treating their initiation and termination as distinct atomic events.

The set of strings representing behaviors of a particular concurrent system is obtained by a two-step process. First, a regular expression, called the *system expression*, is derived from a description of the system in some notation such as a design or programming language. The language of the system expression includes strings representing all possible behaviors of the system. It may, however, also include strings that do not represent possible behaviors, as the system expression does not encode the full semantics of the system description. This language is then "filtered" to remove such strings, using other expressions, called *constraints*, which are also derived from the original system description. A string survives this filtering process if its projections on the alphabets of the constraints lie in the languages of the constraints. The constraints (which need not be regular) enforce those aspects of the semantics of the design or programming language, such as the appropriate synchronization of rendezvous between different tasks or the consistent use of data, that are not captured in the system expression. The reasons for this two-step process, which might not seem as straightforward as generating behaviors directly from a single expression, are discussed in [10].

Our main constrained expression analysis techniques require that questions about the behavior of a concurrent system be formulated in terms of whether a particular event symbol,

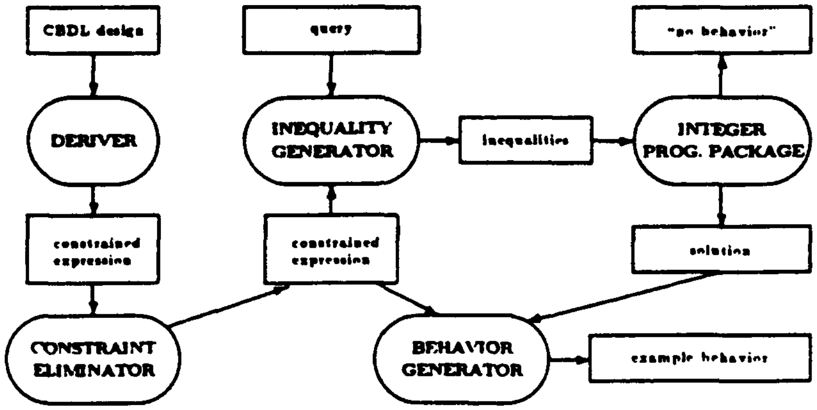


Figure 1: Diagram of Constrained Expression Toolset

or pattern of event symbols, occurs in a string representing a possible behavior of the system. For example, questions about whether the system can deadlock might be phrased in terms of the occurrence of symbols representing the starvation of component processes of the system.

Starting from the assumption that the specified symbol, or pattern of symbols, does occur in such a string, we use the form of the system expression and the constraints to generate inequalities involving the numbers of occurrences of various event symbols in segments of the string. If the system of inequalities thus generated is inconsistent, the original assumption is incorrect and the specified symbol or pattern of symbols does not occur in a string corresponding to a behavior of the system. If the inequalities are consistent, we use them in attempting to construct a string containing the specified pattern.

4.2 The Constrained Expression Tools

After manually applying the constrained expression analysis techniques to a number of small examples with encouraging results (e.g., [1], [4], [5], [16]), we began to construct prototype tools automating various aspects of the analysis. The prototype toolset (see Figure 1) consists of five major components: a *deriver* that produces constrained expression representations from concurrent system designs in a particular design language; a *constraint eliminator* that replaces a constrained expression with an equivalent one involving fewer constraints; an *inequality generator* that generates a system of inequalities from the constrained expression representation of a concurrent system; an *integer programming package* for determining whether this system of inequalities is consistent or inconsistent, and, if the system is consistent, for finding a solution with appropriate properties; and a *behavior generator* that uses the constrained expression and the solution found by the integer programming package (when the inequalities are consistent) to produce a string of event symbols corresponding to a system behavior with the desired properties. The organization of the toolset is illustrated in the figure.

The current toolset is intended for use with designs written in the Ada-based design language CEDL (Constrained Expression Design Language) [9]. CEDL focuses on the expression of communication and synchronization among the tasks in a distributed system, and language

features not related to concurrency are kept to a minimum. Thus, for example, data types are limited, but most of the Ada control-flow constructs have correspondents in CEDL. The deriver in the CEDL toolset is written in Ada, and was developed using Arcadia-produced versions of standard compiler construction tools plus PGRAPHITE. The constraint eliminator, the inequality generator, and the behavior generator are all written in Common Lisp, while the integer programming component, which is built on top of the MINOS optimization package [14], is implemented in FORTRAN. As mentioned in the previous section, we have used our SLI techniques and prototype tools to implement interoperability between the Ada and Common Lisp components of the toolset.

4.3 Experiments with the Toolset

We have begun to use the prototype toolset in the analysis of concurrent systems. The preliminary experiments reported here represent an initial attempt to determine the practical limitations of automated support for the constrained expression approach to analysis. A number of variations of four different systems have been analyzed. First, a standard formulation of the dining philosophers problem provides a basis for comparison with other analysis techniques because of its widespread use as a benchmark problem. The addition of a host task controlling entry to the dining room indicates how the introduction of intra-task dataflow affects tool performance. The readers/writers problem requires analysis of more complex data flow patterns. Finally, we analyze an automated gas station example in which both the synchronization patterns and intra-task dataflow are relatively complex. Varying the number of philosophers in the dining philosophers problems indicates how increasing the number of tasks in the system being analyzed affects tool performance.

The table in Figure 2 gives CPU times for the application of the components of the toolset to these systems. All times are in CPU seconds on a Sun 3/60, except for the deriver time in the DPH-14 case, where memory limitations forced us to run on a 3/260. The first section of the table, with systems labeled DP-n, gives data for analyses of the standard dining philosophers problem involving n philosophers. A bug in the behavior generator prevents us from completing the analysis of the DP-10 case. The next section, with systems labeled DPH-n, gives data for analyses of versions of the problem with n philosophers and a host task that prevents deadlock by limiting the number of philosophers in the dining room. The third section gives the results for two versions, one incorrect and one correct, of the readers/writers example [6]. In the correct version, the analysis determines whether an error flag, representing a violation of the appropriate mutual exclusion, is ever set. The symbol representing the setting of this flag is correctly eliminated by the constraint eliminator; once this is recognized by the inequality generator, no further analysis is necessary. Times for analyses of two versions of Helmbold and Luckham's automated gas station [11], one with a potential deadlock and the other without, are given in the last section. More information on these systems and the analyses (but with less current performance data) is provided in [2].

These initial experiments with the prototype constrained expression toolset are encouraging. The toolset provides complete automated analysis of a range of standard concurrent system examples. Even the prototype versions of the tools are efficient enough to be useful to software developers on examples of moderate size. Unlike the standard approaches to concurrency analysis, which are based on a reachability tree construction that grows exponentially

system	deriver	constraint eliminator	inequality generator	int. prog. (IMINOS)	behavior generator	total CPU time
DP-3	74	1	9	2	19	105
DP-4	82	2	11	3	26	124
DP-5	94	3	14	3	32	146
DP-6	109	4	17	4	38	172
DP-8	142	7	24	5	54	232
DP-10	177	11	30	7		
DPH-3	123	6	16	3	—	148
DPH-4	133	9	22	4	—	168
DPH-5	152	14	30	6	—	202
DPH-6	174	19	38	7	—	238
DPH-8	207	31	57	11	—	306
DPH-10	250	49	82	30	—	411
DPH-14	233	101	135	57	—	526
RW-I	43	6	9	7	124	189
RW-C	63	10	2	—	—	75
GAS-I	75	21	30	13	721	860
GAS-C	76	16	23	13	—	128

Figure 2: CPU times, in seconds, for the constrained expression tools.

with the number of concurrent tasks being analyzed, our toolset does not appear to suffer from exponential performance degradation as problem size increases. Furthermore, earlier experiments show that the constrained expression approach can detect a variety of errors and can be used with a broad range of design notations and programming languages.

We are currently reimplementing the behavior generator, to remove the bug that we have encountered in the DP-10 case, to enable it to use all the information provided by the solution to the system of inequalities, and to add some additional functionality. We expect that significant improvements in its performance will result from the use of more information from the solution.

While improving the prototype toolset, we have also begun to explore additional applications for constrained expression analysis, some of which may lead to enhancements to the underlying formalism and further modifications to the tools. In particular, we have begun to study the application of the constrained expression approach to various scheduling and real-time problems [3].

5 SUMMARY AND CONCLUSIONS

In this paper we have provided brief overviews of our work on two important aspects of automated support for development and evolution of complex software systems. Our work on type models, persistence and interoperability is representative of the directions that object management capabilities for environments are likely to take over the next decade. Our constrained expression toolset is an example of the automated analysis techniques that will be required if such environments are to adequately support development and evolution of

concurrent, real-time or other classes of complex software.

Perhaps the most fundamental aspects of our object management work are its emphasis on strong, abstract typing, and its tendency toward incorporation of object management capabilities (e.g., persistence) into the language in which tools are implemented, as opposed to the current practice of providing such capabilities via a database or file system. We believe that these directions will be important factors in increasing integration, extensibility and interoperability in the next generation of software development environments.

While we believe that the results of our experiments with our prototype constrained expression toolset are significant, we see the empirical approach to evaluation of proposed approaches to software analysis that the experiments represent as equally important. Only by carrying out such empirical evaluations, preferably using a suite of standard examples, can the relative strengths and weaknesses of various proposed approaches be determined. Assembling collections of tools with known, complementary capabilities on top of infrastructures that facilitate their integration and interoperability is the most promising approach to providing automated support for development and evolution of complex software systems.

Acknowledgments

This work was supported in part by NSF grant CCR-87-04478 with cooperation from DARPA (ARPA order 6104), by NSF grant CCR-8806970 and by ONR grant N00014-89-J-1064. Alexander Wolf has played a major role in all of the object management research described in this paper. Lori Clarke, William Rosenblatt and Peri Tarr have also made important contributions to the SDL object management work reported here. Many others of our Arcadia colleagues have provided valuable input to this work. George Avrunin and Laura Dillon have been centrally involved in the development of the Constrained Expression approach to analysis of concurrent software systems. Susan Avery, Ugo Buy, James Corbett, Michael Greenberg, RenHung Hwang, and George Walden have played important roles in the development of the toolset and the experimental evaluation reported in this paper.

REFERENCES

[1] G. S. Avrunin. Experiments in constrained expression analysis. Technical Report 87-125, Department of Computer and Information Science, University of Massachusetts, Amherst, November 1987.

[2] G. S. Avrunin, L. K. Dillon, and J. C. Wileden. Experiments with Automated Constrained Expression Analysis of Concurrent Software Systems. In *Proceedings TAV3-SIGSOFT89: Third Testing, Analysis and Verification Symposium*, pages 124–130, December 1989.

[3] G. S. Avrunin, L. K. Dillon, and J. C. Wileden. Constrained expression analysis of real-time systems. Technical Report 89-50, Department of Computer and Information Science, University of Massachusetts, 1989.

[4] G. S. Avrunin, L. K. Dillon, J. C. Wileden, and W. E. Riddle. Constrained expressions: Adding analysis capabilities to design methods for concurrent software systems. *IEEE Trans. Softw. Eng.*, SE-12(2):278–292, 1986.

[5] G. S. Avrunin and J. C. Wileden. Describing and analyzing distributed software system designs. *ACM Trans. Prog. Lang. Syst.*, 7(3):380–403, July 1985.

[6] R. H. Carver and K.-C. Tai. Detection of synchronization errors in concurrent software by semantics-based analysis. Preprint, 1988.

[7] L. A. Clarke, J. C. Wileden, and A. L. Wolf. GRAPHITE: A meta-tool for Ada environment development. In *Proceedings of 2nd International Conference on Ada Applications and Environments*, pages 81–90, April 1986.

[8] L. K. Dillon. *Analysis of Distributed Systems Using Constrained Expressions.* PhD thesis, University of Massachusetts, Amherst, 1984.

[9] L. K. Dillon. Overview of the constrained expression design language. Technical Report TRCS86-21, Department of Computer Science, University of California, Santa Barbara, October 1986.

[10] L. K. Dillon, G. S. Avrunin, and J. C. Wileden. Constrained expressions: Toward broad applicability of analysis methods for distributed software systems. *ACM Trans. Prog. Lang. Syst.*, 10(3):374–402, July 1988.

[11] D. Helmbold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.

[12] J. E. B. Moss and S. Sinofsky. Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface. In *Proceeding of the Second International Workshop on Object Oriented Data Bases*, Springer-Verlag, September 1988.

[13] W. R. Rosenblatt, J. C. Wileden, and A. L. Wolf. OROS: Toward a Type Model for Software Development Environments. In *Proceedings OOPSLA '89: Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 297–304, October 1989.

[14] M. A. Saunders. MINOS system manual. Technical Report SOL 77-31, Stanford University, Department of Operations Research, 1977.

[15] R. N. Taylor, F. C. Belz, L. A. Clarke, L. J. Osterweil, R. W. Selby, J. C. Wileden, A. L. Wolf, and M. Young. Foundations for the Arcadia environment architecture. In *Proceedings SIGSOFT '88: Third Symposium on Software Development Environments*, pages 1–13, December 1988.

[16] J. C. Wileden and G. S. Avrunin. Toward automating analysis support for developers of distributed software. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 350–357. IEEE Computer Society Press, June 1988.

[17] J. C. Wileden, A. L. Wolf, C. D. Fisher, and P. L. Tarr. PGRAPHITE: An Experiment in Persistent Typed Object Management. In *Proceedings SIGSOFT '88: Third Symposium on Software Development Environments*, pages 130–142, December 1988.

[18] J. C. Wileden, A. L. Wolf, W. R. Rosenblatt, and P. L. Tarr. Specification level interoperability. In *Proceedings of the Twelfth International Conference on Software Engineering*, pages 74–85, March 1990.

[19] J. C. Wileden, P. L. Tarr, and L. A. Clarke. Extending and Limiting PGRAPHITE-style Persistence. Submitted.

REPORT DOCUMENTATION PAGE

1. Recipient's Reference:	2. Further Reference:
3. Originator's Reference: AC/243(Panel 11)TP/1	4. Security Classification: UNCLASSIFIED/UNLIMITED 5. Date: 15.04.91 — 6. Total Pages: 10

7. Title (NU):

Object-Oriented Languages for Interoperability and Incremental Development of Command and Control Information Systems

8. Presented at:

AC/243(Panel 11) Symposium on Military Information Systems Engineering - RSRE, Malvern, UK - 8-10 May 1990

9. Author's/Editor's:
Alain M. Bories

10. Author(s)/Editor(s) Address:
ALCATEL ISR
523 Terrasses de l'Agora
F-91034 Evry Cedex
France

11. NATO Staff Point of Contact:
Defence Research Section
NATO Headquarters
B-1110 Brussels
Belgium
(Not a Distribution Centre)

13. Keywords/Descriptors:

OBJECT-ORIENTED LANGUAGES, INCREMENTAL DEVELOPMENT, INTER-OPERABILITY, COMMAND AND CONTROL INFORMATION SYSTEMS

14. Abstract:

Traditional software development methodologies are not well suited for Command and Control Information systems. Main drawbacks are the 5 to 10 year-usual time lag between the assessment of operational needs and the delivery of the system, the rapid obsolence of information technologies and the constant evolution of the users' needs. The result often is the delivery of a system which does not meet the current requirements, even if, in the best cases, it meets the initial requirements provided in the specifications. A proposed way of overcoming such drawbacks is to build the system in an incremental fashion. The use of object-oriented languages is a key feature to successfully address this issue.

OBJECT-ORIENTED LANGUAGES FOR INTEROPERABILITY AND INCREMENTAL DEVELOPMENT OF COMMAND AND CONTROL INFORMATION SYSTEMS

Alain M. BORIES

* Deputy Director for Marketing and Strategy, Alcatel ISR

This paper begins with an historical overview of past failures in CCIS. It then overviews the characteristics of CCIS. After a brief description of object-oriented languages, it explains how and why the use of these languages solve many problems for CCIS development.

I - HISTORICAL OVERVIEW

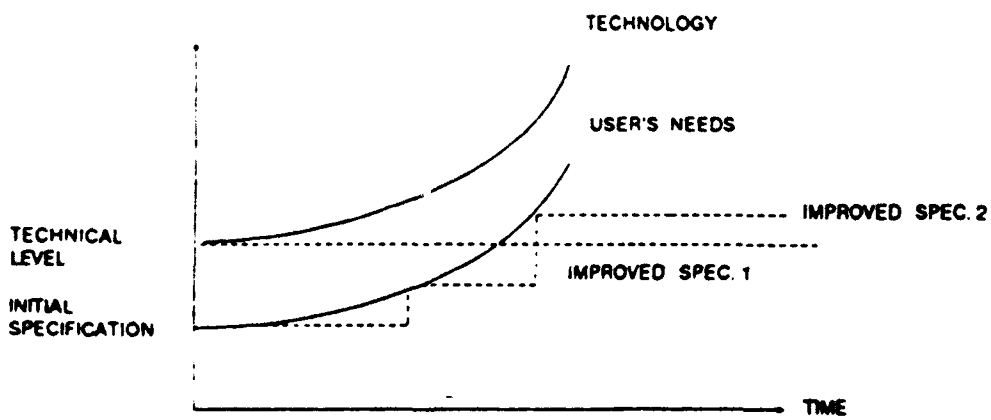
Many failures have occured in the past concerning CCIS. What has been seen are :

. delays in the delivery of the system ;
. obsolescence, meaning that the techniques used are out of date whe the system is delivered ;
. non-compliance, not with the initial specs, but with what the user thinks his system might do.

These facts are not independant. They result mainly from the 5 to 10 years time lag between the assessment of operational needs and the delivery of the system. This is not compatible with the rapid obsolescence of information techniques, coupled with the constant evolution of the user's needs and changes in his organization and procedures. For example, if it is suddenly being decided that Air Defense and Tactical Forces will be managed by a combined Air operations center, it will not change the radars, but it will dramatically change the CCIS.

As can be seen in figure 1, the initial specification is done with some technical margin. The problem is that this margin decreases as the user's need evolves. If nothing is done to adapt the specs to the user's needs, it comes out an obsolete system which does not fulfill the user's needs and which will not be used even though it is compliant with the initial spec. If something is done to adapt it to the user's needs, it may come to a point where the technical level of the initial system is not high enough to comply with the new specs.

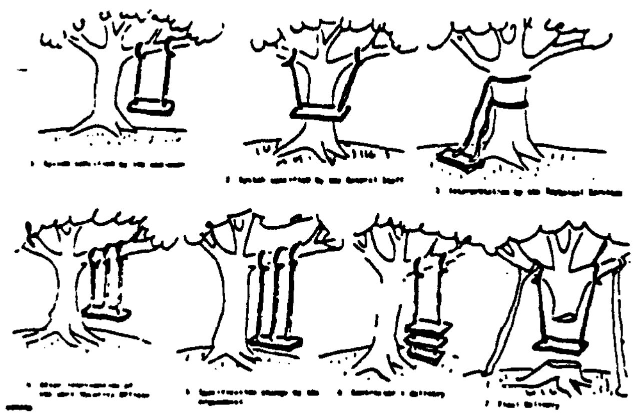
Figure 1 : System evolution



Another reason for failures is the lack of visibility of traditional methodologies (paper specifications) for the end user due to the different interpretations of natural language words. Figure 2 is a humoristic illustration of the fact.

Figure 2 :

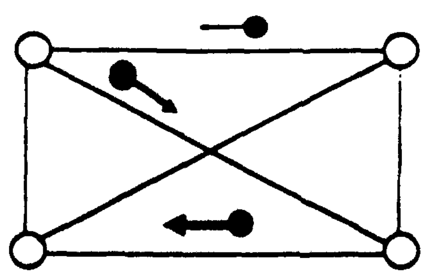
Misunderstandings



II - COMMAND AND CONTROL INFORMATION SYSTEMES (CCIS)

What is a CCIS ? It may be described as a network of decisions centers, which may be called agents : they may be automatic processes, staff brainstorming, single human beings, etc... Each of them receives, processes and dispatches pieces of information. The process involved in each decision center adds value to the initial piece of information as shown on figure 3.

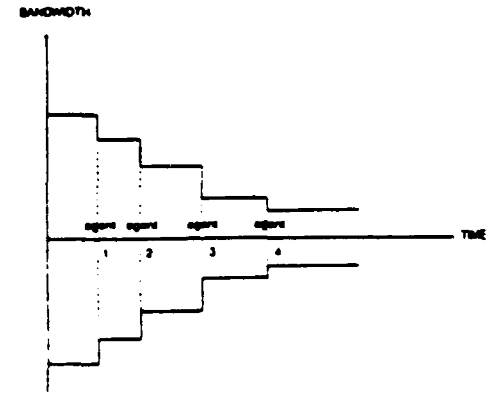
Figure 3 : Agregation of information



The whole process allows to make information more agregated and useable for decision because the bandwidth of the piece of information is reduced as each agent adds value to it (figure 4).

Figure 4 :

Fusion of information



The main characteristics of CCIS are :

- the man -in-the-loop concept : man, that is to say the operator, has to be taken into account in the design of the system, because he is involved in the decision process. It is not just a matter of man-machine interface, where man is outside but tries to interact with the system the best way he can. In CCIS, man is part of the system and is a constraint.

- interoperability : that means that data handled by each agent are to be homogeneous throughout the whole system, and the procedures are to be compatible. It is not just a matter of having the same communications protocols ; but really the same understanding of the data and processes.

- evolutivity : as have been seen, a main reason for failure is the lack of evolutivity. The system has to be designed in order to be able to handle changes in needs, in procedures or in techniques.

- security : that includes confidentiality, but also data integrity and reliability which is a sine qua non condition to the process of decision - making through such a system. And also testability, that is to say the ability to check that the system is doing what it is supposed to do.

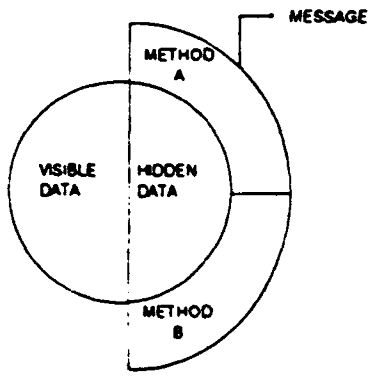
III - <u>OBJECT-ORIENTED LANGUAGES (OOL)</u>

Before going on on CCIS, this paragraph will give a very short description of object-oriented languages, to be able to show their use in CCIS. And first, what is an object ? An object is a representation of a real-world entity that is an encapsulation of visible data and methods with hidden (or protected) data structure. Each object may

send messages to other objects to invoke an operation (when this operation is visible) in another object. An object receiving a message will activate the method corresponding to it (figure 5).

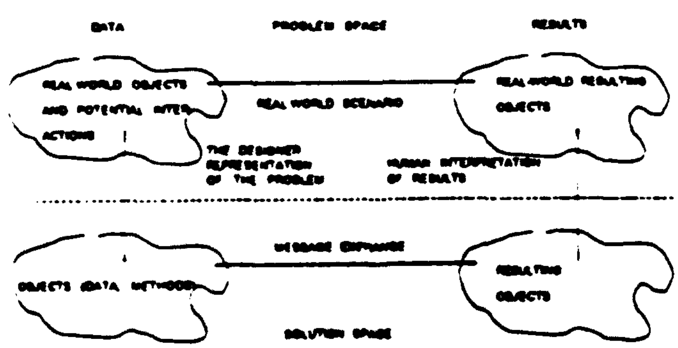
Figure 5

Object structure



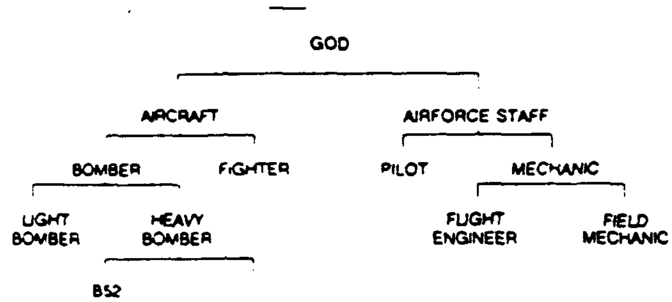
In a more practical way, the representation of the real world may be depicted as shown on figure 6, with its modelization through objects. Scenarios in the real world are message exchanges in what may be called "the object-oriented world".

Figure 6 : Object-oriented approach to represent the real world



Objects belong to classes and sub-classes which are themselves objects. Characteristics are inherited from upper-level objects. This is particularly useful to be able not to repeat all the characteristics which are derived from the class to which the object is belonging. Figure 7 shows an example of the object description of a system.GOD is the uppermost class. It means General Object Description (!).

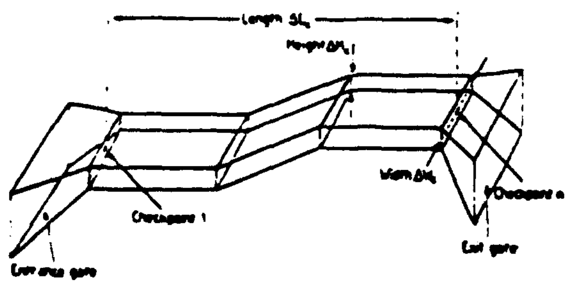
Figure 7 : Principles of inheritance



Each object is described by its name, the class to which it belongs, its characteristics, visible or not, the methods which may be applied either to external or to internal data, its visibility to or from other objects, and the exceptions allowing to assess error conditions.

An example is shown on figure 8 : an airspace corridor, which may be used for an airspace management system. This corridor is composed of several objects, for example, a parallelepiped with some characteristics : length, width and so on. One method embedded in the object "corridor" is the computation of the total length. More interesting is the way the corridor is built during the design process. It interacts through messages with other objects such as terrain objects, other corridors, regulations (which are an object consisting of several methods) and so on. It will interact also with the object 'aircraft", allowing or not an aircraft to be in the corridor space, depending on its occupation. We can go further and further and build a whole airspace management system like that.

Figure 8
Example of an object

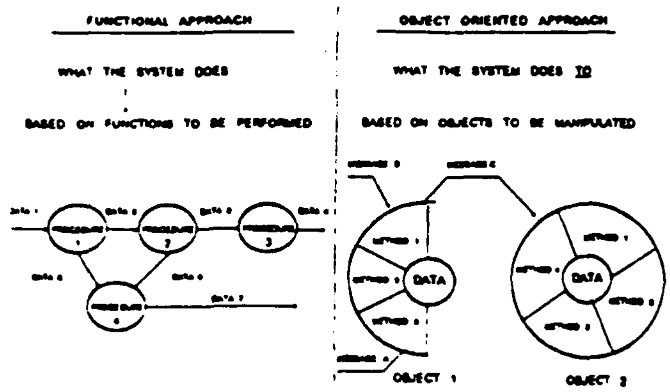


Typical spatial safe corridor with 1 level change

To summarize the differences between traditional languages and object-oriented languages, figure 9 shows the functional approach with data processed by procedures, and the object oriented approach where objects are sending messages, data and methods being embedded in the objects.

Figure 9

Functional vs. Object-oriented approaches.



IV - CCIS AND OOL

This paragraph will try to show why and how object-oriented approaches may answer the problems of CCIS development which have been emphasized before. Following are the main characteristics of CCIS, and why object-oriented languages are well-suited to them.

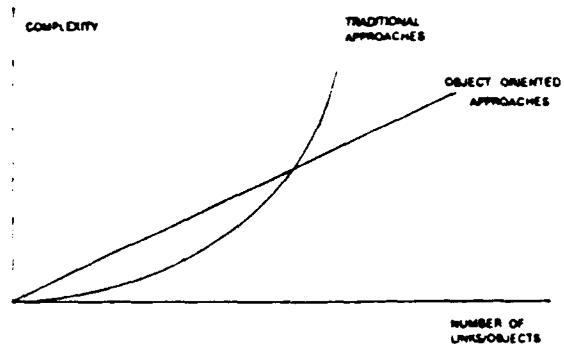
- first, the man-in-the-loop concept : the use of rapid prototyping has often been emphasized to deal with this characteristic because it allows frequent checking of the exact users'needs. Object-oriented languages are well-suited for this because they have the key advantage of being a representation of the real world, easy to build by a medium intelligent user, so to speak. They reflect the process of designing CCIS with very little need of technical expertise.

- interoperability : this is embedded in the description of objects. All users are sharing the same objects with the same characteristics and data, which are therefore described as common entities with well-agreed behaviour.

- evolutivity : as have been seen, it is very easy to create new classes or new objects by instanciation of an existing class, or modify existing objects by adding new characteristics and methods. This what may be called "granularity" (rather than modularity) enables the designer to start from simplified objects or a reduced set of objects, thus allowing incremental development. This is essential for dealing with the problem of the time lag between specification and delivery of the system. The prototype may be constantly improved and put into operational use as soon as it provides significant improvement to the former version. Evolutivity is also enhanced by the simplification of the problem. Usually

complexity increases exponentially with the number of links between data and processes in conventional development methodologies. With object-oriented languages, granularity and communication between objects through the exchange of messages gives a quite linear increase of complexity, as shown on figure 10.

Figure 10



- at last, security : it is handled by the notion of hidden data, each object being allowed to see only some parts of another object. The operator is a particular object and is cleared to access to certain parts of the objects, data and/or methods.

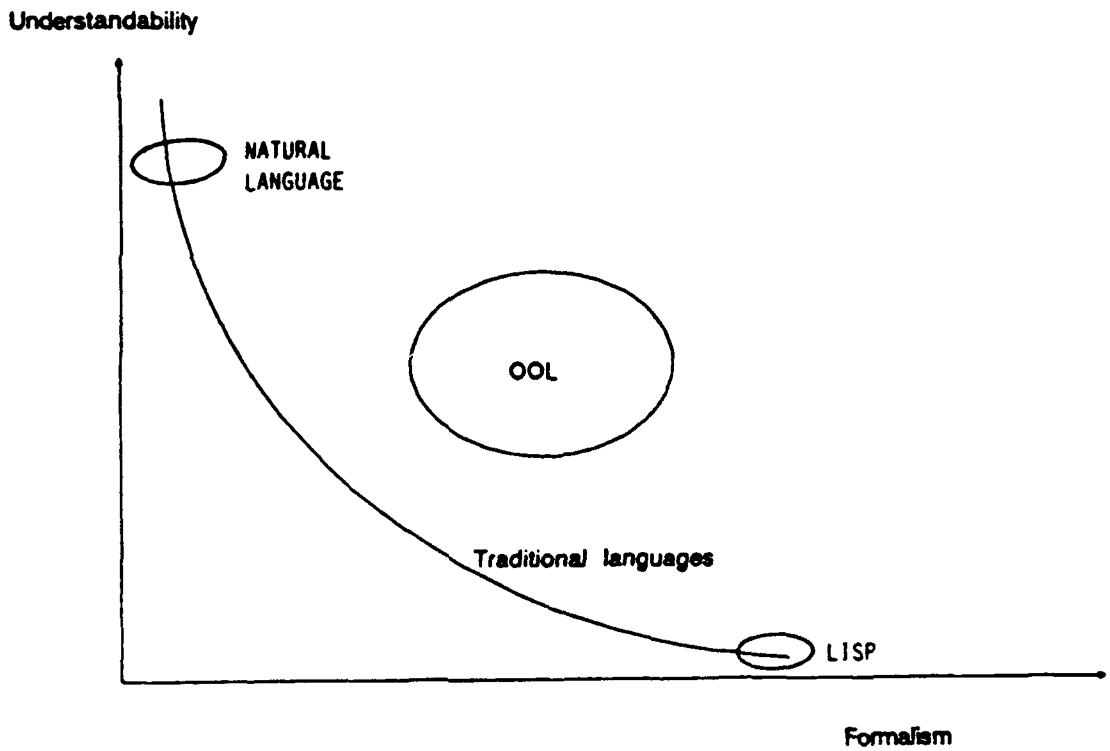
The integrity of data was also mentioned before : this is ensured by what may be called the "packaging" of data and the notion of inheritance : a change in a characteristic or a method is inherited by lower classes objects and checked by the object-oriented environment itself.

As far as reliability of data is concerned, all users have the same defintion of data and methods, and "plausibility" may be set as a characteristic of a given object, which will have the same meaning for the whole system and which will be processed and combined with other plausibilities by the mean of methods.

As far as testability is concerned, it is handled through the granularity of the object-oriented approach which enables an automatical non-regression process.

At last, the problem of the understandability of the specifications is better handled through OOL, as shown on figure 11, than with traditional languages where the more it is formalized, the less it may be understood.

Figure 11

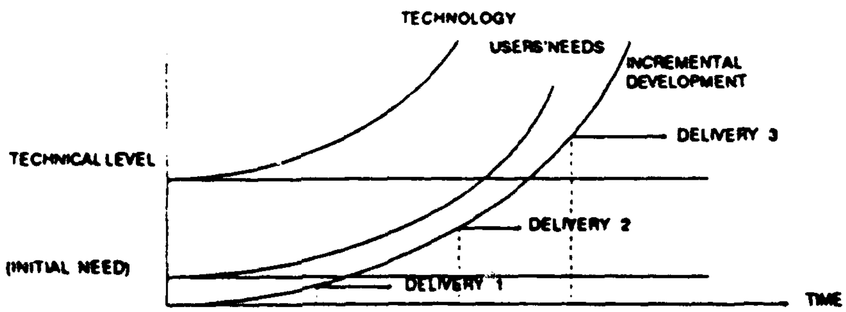


V - CONCLUSION

As have been emphasized before, CCIS cannot be handled by conventional development methodologies, where development is not begun until the specs are settled and approved. To come back to figure 1, incremental development with the involvement of the end-user during the whole cycle is a key issue which can only be coped with, at the time being, through object-oriented approaches. This allows to follow much more closely the curves of evolutions of users'needs and technological trends as shown on figure 12. But this will probably put into question, in the very next future, the traditional procurement procedures for such systems because of the difficulty of controlling the costs. This is a challenging question, but some elements can be given for answering it :

- software is expensive. The granularity of object-oriented approach allows to spare time and money thanks to reuseability;
- maintenance is much easier, also because of granularity, and thus much cheaper;
- and the customer does not end with a system which is not used operationnally. Which is an interesting way of saving money.

Figure 12



NATO UNCLASSIFIED

REPORT DOCUMENTATION PAGE

1. Recipient's Reference:	2. Further Reference:
3. Originator's Reference: AC/243(Panel 11)TP/1	4. Security Classification: UNCLASSIFIED/UNLIMITED 5. Date: 15.04.91 — 6. Total Pages: 12
7. Title (NU): A Structure for Distributed Command and Control Information Systems using Commercially available Software	
8. Presented at: AC/243(Panel 11) Symposium on Military Information Systems Engineering - RSRE, Malvern, UK - 8-10 May 1990	
9. Author's/Editor's: Dr. Werner Storz	
10. Author(s)/Editor(s) Address: FGAN/FFM/RuF Neuenahrerstr 20 D-5307 Wachtberg Germany	11. NATO Staff Point of Contact: Defence Research Section NATO Headquarters B-1110 Brussels Belgium (Not a Distribution Centre)
12. Distribution Statement: Approved for public release. Distribution of this document is unlimited, and is not controlled by NATO policies or security regulations.	
13. Keywords/Descriptors: ISO/OSI REFERENCE MODEL, CCIS, COTS, ADA, DISTRIBUTION, SECURITY	

14. Abstract:

The implementation of modern distributed C&C information systems (CCIS) should follow three principles:

- The structure and the services of the ISO/OSI Reference Model should be an integral part of the systems

- The programming language Ada should be applied

- Commercial-off-the-shelf (COTS) should be used as much as possible.

The paper will discuss problems and solutions for development following these lines.

NATO UNCLASSIFIED

A STRUCTURE FOR DISTRIBUTED COMMAND AND CONTROL INFORMATION SYSTEMS USING COMMERCIALLY AVAILABLE SOFTWARE

Dr. Werner Storz *

* Chief C&C Division at FGAN/FFM,
D 5307 Wachtberg-Werthhoven

1. INTRODUCTION

Experience with existing Command and Control (C&C) information systems has shown that these systems have an extended life cycle of about 20-30 years. Planning C&C information systems such time periods have to be taken into account. Within these time periods, however, the political, military and technical situation will change with the result of changing requirements. The actual political development in Europe supports this statement. It is therefore necessary to design C&C information systems for adaptibility. In this way they might be adapted to changing operational and technical requirements through their life cycle.

To get this flexibility we have to design a well-structured modular system with well defined interrelationships and common interfaces between the modules. The adaptation of the system will then be done by (local) modification or change of modules. This procedure is also suitable to benefit from development of commercially available software which might be integrated into the C&C information system as new modules.

Another aspect of influence on the design of C&C information systems is the requirement of cooperation between deployed C&C information systems. This is especially important in the context of NATO. In the Alliance C&C information systems of distinct hierarchical levels and different nationalities have to be interoperable. To guarantee interoperability for such distributed, heterogeneous systems standards for the communication protocols are imperative.

Bearing in mind these requirements we want to define an architecture for distributed C&C information systems. There is already ongoing work in several NATO-committees and working groups concerning this problem area (protocols: AC/302, Subgroup 9; Ada: AC/302, (ADA)). We will pay attention to results of these groups in following three principles concerning the structures of distributed C&C information systems:

- The structures and the services of the ISO/OSI Reference Model should be an integral part of the system.
- The programming language Ada should be applied.
- Commercial of the shelf (COTS) software should be used as much as possible.

We will now discuss problems and solutions for the development of C&C information systems following these lines.

2. PROTOCOL STACKS IN THE ISO/OSI REFERENCE MODEL

The common structure of the OSI Model with its seven layers is shown in Figure 1. This evidently well structured model [1], [2] does not show its inherent weakness: In each layer several alternative protocols are available. Especially in the military field different protocols in the lower layers will be needed and used for different networks: Wide Area Networks (e.g. X.25, ISDN), Local Area Networks (e.g. Ethernet, FDDI) and other types of network offered by the ever evolving technology.

Application
Presentation
Session
Transport
Network
Link
Physical

Figure 1: Open Systems Interconnection Model

The OSI model offers eight possible addressing schemes for the network addressing and we may choose between five transport protocols at the transport level. At the higher protocol layers we have also the possibility to select different services.

Not all combinations of protocols result in a functioning communications system. For this reason profiles of protocol stacks have been defined. But these profiles are only partially compatible. That's why one should select one specific profile for a network of communicating C&C information systems. Figure 2 shows such a profile. The layer 3c contains the internetwork protocol CLNP (Connectionless Network Protocol). The subnetworks (sublayers of layer 3) might be X.25-networks or other networks (e.g. ISDN, Ethernet, FDDI).

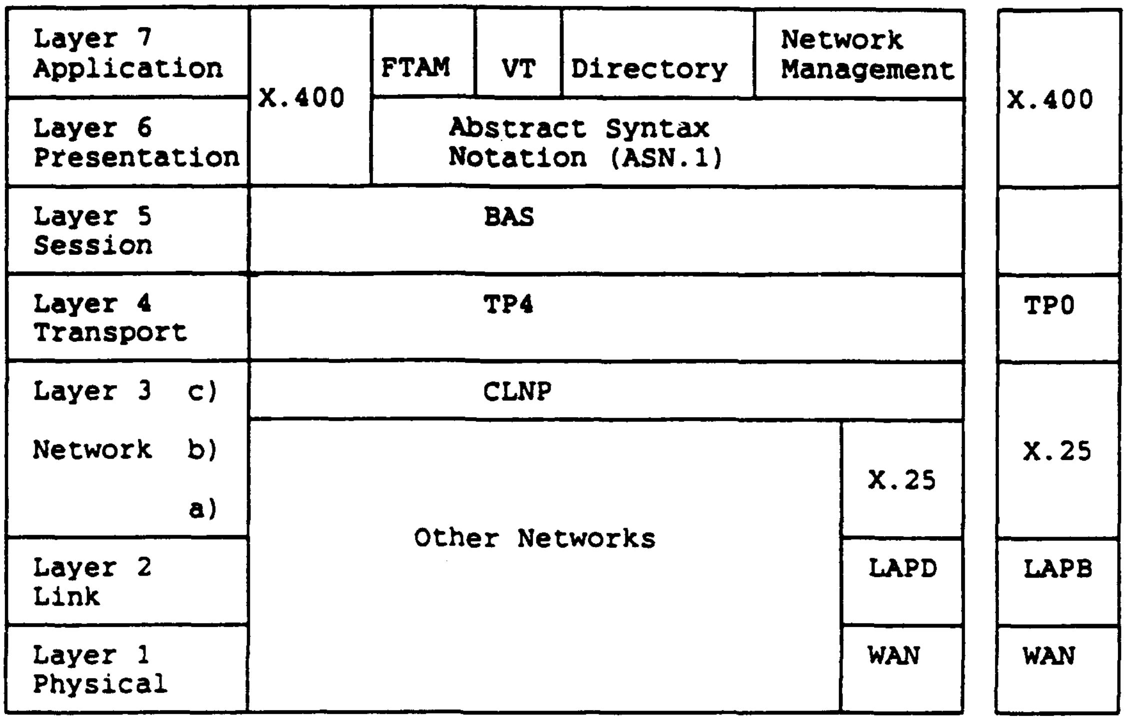


Figure 2: OSI Protocol Suite

Thus on top of heterogeneous subnetworks a common internetwork protocol is available. The internetwork protocol is connectionless. We therefore need the connection-oriented transport protocol TP4 on top of it. The protocol layers 5, 6, 7 need no further discussion in this context. A second protocol stack shown here is X.400-TPO-X.25. This profile might be advantageous for using commercially available networks to transport X.400 mail. The profiles shown in Figure 2 are roughly equivalent to that defined by GOSIP (Government Open System Interconnection Profile) for the usage in U.S. governmental systems [3].

The above mentioned problem of a common addressing scheme in the internetwork sublayer (Network Service Access Point (NSAP) address) may be solved in the following way: The heterogeneous subnetworks use different address structures. For a general structure a hierarchical address structure seems to be adequate. The OSI model has two hierarchical addressing schemes. Examples of both are presented in Figure 3. The upper part of Figure 3 shows the NSAP address of GOSIP. The Authority and Format Identifier (AFI) of this type of NSAP address has the value 47. The Initial Domain Identifier (IDI) with the value 0006 specifies the address as an U.S. governmental one. The

AFI	IDI	DSP						
47	0006	Vers	AdminAuth	Res	RoutDom	Area	EndSystem	Nsel
1	2	1	3	2	2	2	6	1

GOSIP-NSAP-address

IDP		DSP		
AFI	IDI	UKDP		UKDSP
39	826	UKFI	UKDI	
1	2	1/2	3/2-7/2	12 - 10 Octetts

BSI-NSAP-address

Figure 3 : Examples of hierarchical OSI-NSAP-Addresses

attribute 'EndSystem' designates the subnetwork address, the attributes 'Administration Authority', 'Routing Domain', 'Area' are the hierarchical part of the address.

In the lower part of Figure 3 the NSAP address is shown as defined by British Standards Institute (BSI)[4]. The Authority and Format Identifier of this type contains the value 39. The Initial Domain Identifier (IDI) contains the Domain Country Code. The value 826 indicates the country UK. The UK Domain Part (UKDP) identifies an organisation which is responsible for the administration of the UK Domain Specific Part (UKDSP) an address part which might be hierarchically structured. Depending on the value of the UK Format Identifier (UKFI) the components UKDP and UKDSP are of variable length. UKDP varies from 2 to 4, UKDSP varies from 12 to 10 octetts. DSP, the sum of both components, has a fixed length of 14 octetts.

To implement a distributed C&C information system on top of heterogeneous networks an NSAP address of the discussed types has to be specified.

3. A LAYERED ARCHITECTURE FOR DISTRIBUTED C&C INFORMATION Systems

We propose a model for a modular distributed C&C information system which uses the protocol stack shown in Figure 2 as standards as well as commercially available software. In this

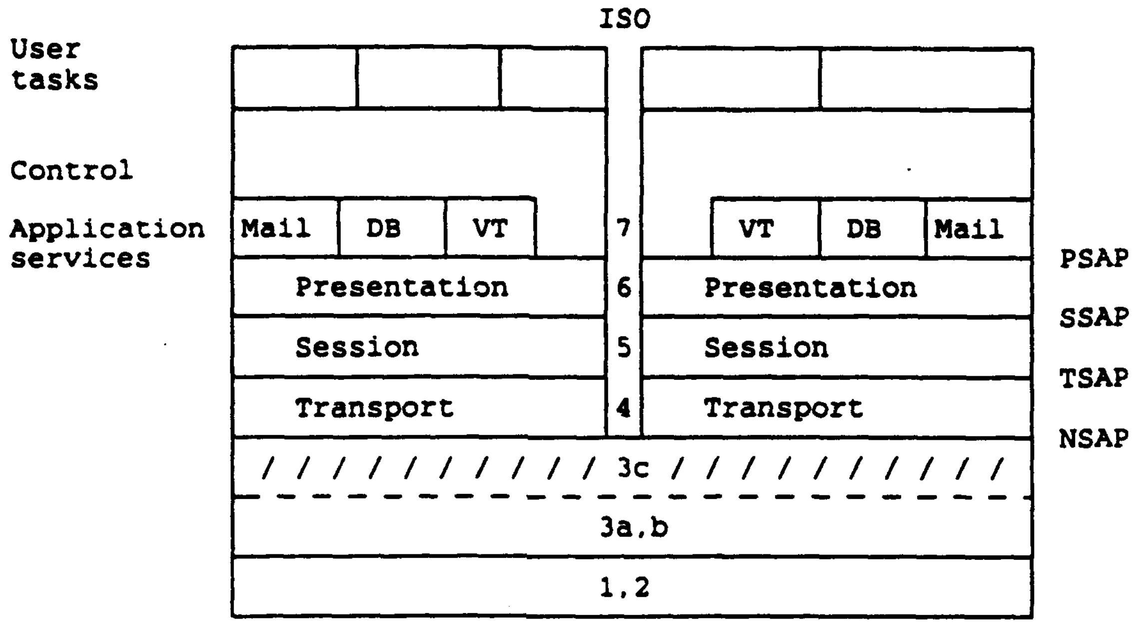


Figure 4: Layers of a Distributed C&C Information System

case the structure of Figure 4 can be assumed as a feasible configuration. In the uppermost sublayer of the application layer user tasks are executed. In course of processing these tasks access application services such as data base, mail, virtual terminal. A control sublayer coordinates these accesses to the different application services.

Similar structures are discussed and proposed in the ATCCIS Working Group. This group meets under the direction of a steering group which is chaired by SHAPE and consists of representatives from the Central Region nations and Allied Forces Central Europe. Its task is to define a system concept for an army tactical C&C information system for the year 2000 and beyond [5], [6].

3.1 Usage of Commercial Software and Ada

Quite a number of commercial software packages may be used on the basis of the structure of Figure 4. The OSI protocol stack for the levels 1 to 6 will soon be available. X.400 implementations exist for the mail service. The data base service is not yet available as OSI-implementation - there is up to now no standard - but Commercial-of-the-shelf (COTS) products (e.g. INGRES, ORACLE, INFORMIX [7],[8],[9]) might be used as servers with a reduced possibility of general distribution. The Virtual Terminal is a standard now, but implementations are not yet available. To overcome this difficulty one might use COTS pro-

ducts based on the X Windows protocol [10], which, in addition, offers more functionality at the user dialog interface. Using this available software a C&C information system may be constructed up to the level of the servers of our model. The interface to the servers, the control process and the user tasks still remain to be implemented.

But a main requirement still needs further consideration to use the programming language Ada as language standard in NATO. A lot of the available software is written in C. These modules cannot fulfill the NATO requirement of using Ada for programming in C&C information systems. But the above mentioned upper layers of the system (interface, control process, user tasks) might be programmed in Ada. An example for such a proceeding is given later.

3.2 Security Problems

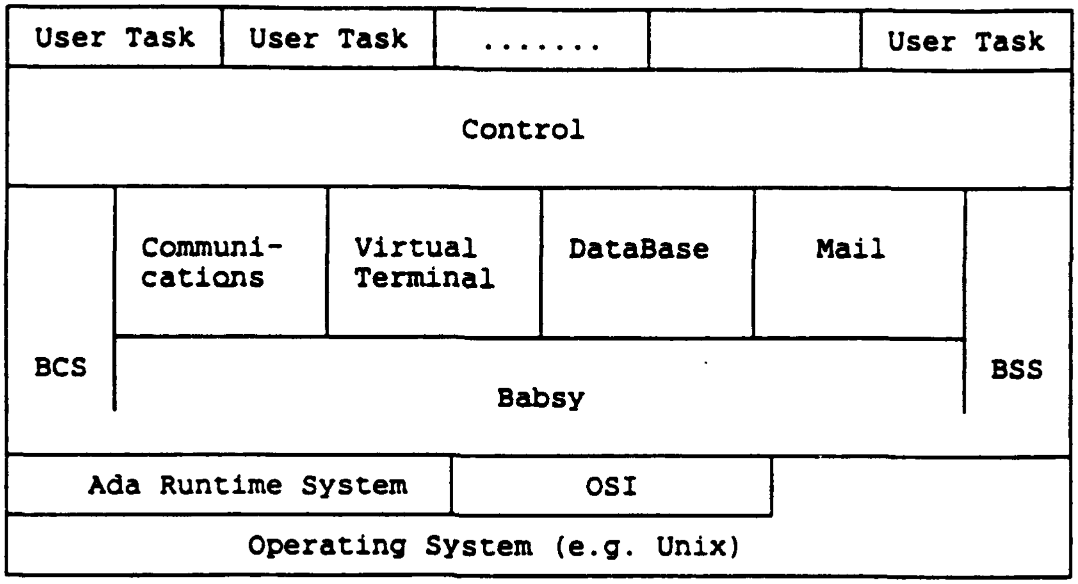
Up to this point we did not discuss the requirements concerning security in C&C information systems. The standardized OSI protocols do not contain the necessary funtions. There is ongoing definition work in AC/302, Subgroup 9. This group proposes to put a security sublayer between layer 3 and 4 to handle the transport oriented communications security tasks [11]. Another sublayer should be introduced at the bottom of the application layer to deal with the application oriented communications security tasks. Protocols with similar functions have been introduced to ISO via the American National Standards Institute. These protocols are now accepted and discussed as work items by ISO.

The standardization and the implementation of these protocols on the commercial market will last some further time. In the meantime there exist two solutions for the communications security problem: implementing own security protocols for the above mentioned sublayers or using closed networks.

With regard to the C&C information system model as shown in Figure 4 further security functions are needed at the interface between control and servers. The security functions belonging to the control layer are oriented to data flow, those belonging to the servers are oriented to data access. For the part of the C&C information systems model where no commercial software is available these functions have to be programmed anyway preferably in Ada as mentioned above.

3.3 An Experimental Implementation

To gather experience in the field of implementing C&C information systems in the structure just discussed a project was started at our institute. The name of the project is EIGER (Experimentelles Informationsystem auf der Grundlage eines



BCS = Basic Communication System
BSS = Basic Security System
Babsy = Basic Operating System

Figure 5: The Structure of EIGER

Rechnernetzes (experimental information system on the basis of a computer network)). We have designed the system in the structure shown in Figure 5. An Ada programme is put on top of the operating system and the OSI protocols. This Ada programme consists of several Ada tasks following the structure of Figure 4. Each box represents an Ada task [12].

The upmost sublayer contains the parallel user tasks. The following sublayer is the control task and the next one contains the tasks interfacing the servers. Babsy is a basic scheduling system for all the Ada tasks. BCS is a basic communications system (Ada task) which allows symmetric communication between all Ada tasks instead of the direct asymmetric rendezvous concept of Ada. BSS, the basic security system is an Ada packet with basic security functions used by the control layer and the servers for the security handling of data flow resp. data access [13].

The Ada tasks interfacing the servers communicate directly with the OSI servers (e.g. mail). If OSI servers are not available, commercial-of-the-shelf-software (COTS) should be used. An example of such a non-OSI but commercially available server is a data base server. Figure 6 shows the integration of a

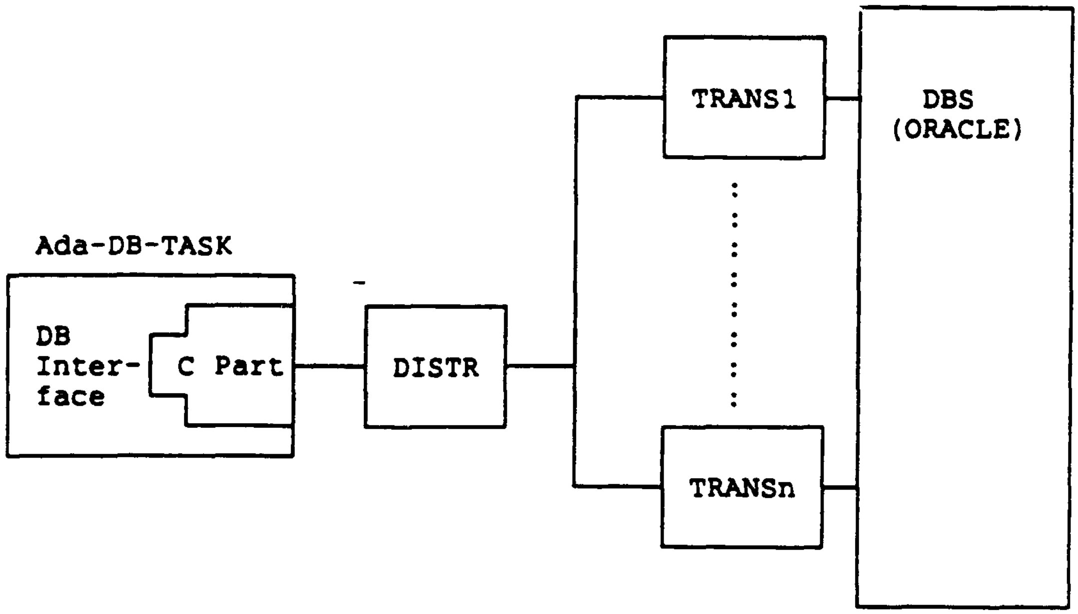


Figure 6: Attachment of a Data Base to EIGER

local data base system as server into EIGER. To provide the needed data base system functionality approved and standardized software should be used. We therefore use the standardized data base language SQL [14] and as an example of an existing relational data base system ORACLE, operating on UNIX.

For the implementation of the interface between the Ada-DB-Task of EIGER and ORACLE two problems have to be solved: The handling of parallel transactions and the transformation of data structures from Ada- to C-programmes and vice versa. To handle parallel transactions ORACLE needs one process for each transaction. The Ada-DB-Task has to control service calls for several parallel transactions. Thus for each transaction a process TRANSi will be created. The process DISTR creates and deletes these processes TRANSi. Considering the problem of data transformation we have to meet the following requirement: Data structures have to be designed in such a way that they can cope with the generality and variety of all possible data base services. Therefore we need dynamic data structures containing the description of basic types and constructed types. Such a data structure exists in ORACLE. In EIGER a descriptive data structure similar to the Abstract Syntax Notation (ASN.1) of ISO has been specified for usage in the Basic Communication System. In the Ada-DB-TASK the transformation of these two descriptive data structures has to be done. Using this process structure and data transformation service calls are sent from the Ada-DB-TASK to the server and results are received the way back.

Without going into further detail we want to state that the last mentioned problem concerning the need of descriptive data at a server interface is a general one because all servers in Figure 5 have to be independent from application data structures.

4. CONCLUSIONS

We showed in the above sections how to use and integrate commercially available software in a C&C information system. Two areas of interest were addressed: OSI protocol stacks up to the application layer (server functions) and non-OSI server modules with commonly used service interfaces. The requirement to use server modules limits the application of COTS software because most COTS software has no interface for service request calls. There exist e.g. commercial office communication systems. The functionality of such a system is needed in a C&C information system. But an office communication system is a closed system with integrated man machine interface, data base system and communications. Normally no service request interfaces are available. It is expensive to split up such a closed system into modules with service interfaces that might be integrated in a C&C information system with a structure following Figure 5. With this restriction in mind COTS software may be used only for data base or man machine interface servers.

We started with the goal to define the architecture of a distributed C&C information system using commercially available software as far as possible. We specified the subrequirements

- use OSI
- use Ada
- use COTS

In Figure 5 a structure was defined that takes into consideration all these points of view: Ada may be used for programmes in the upmost layer. In the lower layers commercial software is available, which in most cases has not been programmed in Ada. In this lower layer OSI implementations will soon be available. But the absence of security features in the OSI protocols will remain a problem in the near future. Servers not available as OSI protocols especially local servers are available as COTS software and may be integrated in the C&C information system.

We have demonstrated that modular, distributed and adaptive C&C information systems using commercially available software are feasible and that this usage of available software may save some funds. But development effort for the implementation of a C&C information system is still needed: The available modules have to be adapted and integrated to build up a system kernel, in the available modules the security components need

further development and the specific C&C-application functions, which make up a big part of the system, have to be designed and implemented.

5. REFERENCES

[1] A.S. Tannenbaum: Computer Networks, Second Edition, Prentice-Hall Eaglewood Cliffs, 1989

[2] J. Henshall, S. Shaw: OSI Explained, end-to-end computer communication standards, Ellis Horwood Ltd. Chichester, 1989

[3] The GOSIP Advanced Requirements Group; U.S. Government Open System Interconnection Profile (GOSIP) Draft, Version 2.0, National Institute of Standards and Technology, Gaithersburg MD 20899, April 1989

[4] Draft British Standard Guide: The UK scheme for the allocation of ISO-DCC format OSI NSAP-address, Version 8, July 1989

[5] K.H. Wagner: Future Interoperability in Army CCIS - Army Command and Control Information System Study (ATCCIS), AFCEA, Oslo Symposium, April 1989

[6] M.R. Krick: The Army Tactical Command and Control Information System (ATCCIS), Technical Report STC TR-120, Nov 1988

[7] M. Stonebraker: The INGRES Papers, Addison-Wesley Publishing Company, 1986

[8] R.B. Bisland: DATABASE MENAGEMENT Developing Application Systems Using ORACLE, Prentice-Hall International Editions, 1989

[9] R. Finkelstein, F. Pascal: SQL Database Management Systems, BYTE, January 1988, P. 111-118

[10] R.W. Scheifler: X Protocol Reference Manual for Version 11, O'Reilly & Associates, Inc., July 1989

[11] NATO OSI Security Architecture "NOSA", AC/302 (SG/9) D/48, Sept 1988

[12] G. Bühler: Implementierungsansätze für ein experimentelles Führungsinformationssystem unter Verwendung der Programmiersprache Ada,
Forschungsbericht 398, Forschungsinstitut für Funk und Mathematik, D-5307 Wachtberg, März 1990

[13] M. Gasser: Building a Secure Computer System,
Van Nostrand Reinhold, New York, 1988

[14] C.J. Date: A Guide to the SQL Standard,
Addison-Wesly Publishing Company, June 1987

NATO UNCLASSIFIED

REPORT DOCUMENTATION PAGE

1. Recipient's Reference:	2. Further Reference:
3. Originator's Reference: AC/243(Panel 11)TP/1	4. Security Classification: UNCLASSIFIED/UNLIMITED 5. Date: 15.04.91 — 6. Total Pages: 14

7. Title (NU):

Army C3I System Software Design: A Case Study

8. Presented at:

AC/243(Panel 11) Symposium on Military Information Systems Engineering - RSRE, Malvern, UK - 8-10 May 1990

9. Author's/Editor's:
(*) IPA P.Y. Simonot - (**) IETA J.L. Auboin

10. Author(s)/Editor(s) Address:	11. NATO Staff Point of Contact:
(*) Defence Research Section NATO HQ B-1110 Brussels (**) DAT/SEFT - Fort d'Issy 92131 Issy-les-Moulineaux France	Defence Research Section NATO Headquarters B-1110 Brussels Belgium (Not a Distribution Centre)

13. Keywords/Descriptors:

C3I SYSTEM, SYSTEM MODELLING, DISTRIBUTED SYSTEM, SOFTWARE DESIGN, INTEROPERABILITY, RECONFIGURABILITY, GRACEFULLY DEGRADABLE SYSTEM, EVOLUTIONARY SYSTEM DESIGN

14. Abstract:

The paper discusses the French Army C3I system which was fielded beginning in 1988. The system was designed primarily as an evolutionary information system. The paper presents the basic concepts and the software architecture used to achieve this goal. Emphasis is also put on specific military features regarding system survivability namely, database distribution and duplication, on-line reconfiguration and Command Post "Step-up".

NATO UNCLASSIFIED

D.2.1 AC/243(Panel 11)TP/1

ARMY C3I SYSTEM SOFTWARE DESIGN: a case study

by

Ingénieur des Etudes et Techniques d'Armement J.L.Auboin(*)
Ingénieur Principal de l'Armement P.Y. Simonot (**)

1. INTRODUCTION
   - 1.1 Background
   - 1.2 Initial System Modeling
   - 1.3 Context of the Full-Scale Development
2. DESCRIPTION OF THE SYSTEM
   - 2.1 General
   - 2.2 Army and Corps Level
   - 2.3 Division Level
   - 2.4 Communications and Interoperability
   - 2.5 Military Issues
3. SOFTWARE GENERAL ARCHITECTURE
   - 3.1 Application Host Software Package Concept
     - 3.1.1 Information Description Language
     - 3.1.2 Software Organization
   - 3.2 Application Host Software Package Design
   - 3.3 Application Software Architecture
4. SPECIFIC FEATURES
   - 4.1 Survivability Requirements and their Refinements
   - 4.2 Duplicated/Distributed Database Management
   - 4.3 Command Post "step-up"
   - 4.4 On-line Configuration/Reconfiguration Management
5. CONCLUSIONS AND RESEARCH RECOMMENDATIONS

(*) Information Processing and Systems Department (Département Informatique et Systèmes) DAT/SEFT - Fort d'Issy-les-Moulineaux - 92131 ISSY-les-MOULINEAUX - FRANCE

(**) Defence Support Division - NATO Headquarters - 1110 - BRUSSELS - BELGIUM (previously Head, Information Processing and Systems Department - DAT/SEFT)

1. INTRODUCTION

1.1 Background

Activities on Army C3I systems started in France early in the sixties together with the development of the French Artillery C3 System (ATILA) and the French Communication system RITA. Two prototypes were developed and fielded respectively in 1968 and 1974. Both were to be used at the Division level; the first one, SERPEL (in 1968) was dedicated to the dissemination and processing of Intelligence information. SYCOMORE which has been fielded from 1974 to 1976 was designed to support the Intelligence and Manoeuvre Division cells.

Following the reorganization of the French Army and the suppression of the Battalion command level, it was agreed that a Command and Control (C2) system should focus on the Corps level and, based on the experience gained from the development of the previous prototypes, a test-bed for a new system named SACRA was developed and then used by a specific operational cell in 1983 and 1984. The test-bed implemented a full Corps Command Post (CP). It was designed as a network of mini-computers and used commercial hardware compatible with military equipments. Two different packet switching networks were used: a datagramme network and a prototype of a dedicated ring network including automatic reconfiguration features. A number of both operational (user) and technical concepts were validated at this stage and alternatives for the full scale development of the system were proposed.

1.2 Initial System Modeling

Key elements of any full scale development decision were the evaluation of the future system performance and the assessment of its feasibility. In this respect, system modeling was extensively used to prepare various architecture options. The modeling process involved three steps:

(1) a functional analysis to define a logical view of the work performed by the users in a Corps CP; each process was analyzed as a succession of elementary operations; the result was a model giving the frequency of each elementary operation performed at any workstation in the CP; the number of instructions to be executed and the number of mass storage accesses required for each of these elementary operations were evaluated using the existing testbed software; this led to an evaluation of the computing power required for each main function in the system; this evaluation was independent of any system design;

(2) hardware component modeling: the system was viewed as a network of hardware components: workstations, computers, mass storage equipments, local network access points, RITA network access points, etc; the elementary operations identified in the logical model (1) were

each component was established using queuing network models; various mappings were envisaged leading to various behavioral models of each component;

(3) overall system performance evaluation; the final step was to map the functional model developed in (1) to a network of hardware components the behaviour of which had been defined during step (2); one of the main objectives was to evaluate the system response time to a user input; the modularity of the modelling approach used in steps (1) and (2) allowed the evaluation of a large number of different architectures.

The main outcomes of this modeling approach were that:

(1) the system could be designed as a network of work-stations and 68000 family-based micro-computers as database and communication processors; minicomputers should be included in the design only if complex simulation and/or computation was to be used to provide decision aids;

(2) the principal limitation to response time was the mass storage access conflicts, which could be avoided by distributing the database and temporary files over a number of devices;

(3) communication processors should be used to interface the CP's to the communication systems (RITA and commercial networks); specific access management policy should be implemented to avoid contention on circuit switching access points.

1.3 Context of the Full-Scale Development

When the decision was made to develop the system, two main constraints were defined:

(1) in order to make the system affordable, the design should use commercially available hardware and software whenever possible; a second generation system would be developed in the mid-90's using ruggedized or fully-militarized hardware as appropriate;

(2) an evolutionary approach should be used with a view to fielding a functionally limited version of the system within 3 years; following versions would provide extended user functions; a pragmatic approach was favoured to allow for user/designer interactions during the design process.

The system (named SICF: "Système Informatique de Commandement des Forces") was developed under contract from the DAT/SEFT by THOMSON/DCS as the main contractor. The system provides information processing support to the 1st Army, Corps and Division CP's.

2. DESCRIPTION OF THE SYSTEM

2.1 General

In order to limit the software development workload it was decided to use a common design for the three command levels to be implemented. This design was deduced from the modeling described above, taking into account specifics imposed by the use of commercially available components. Basically, each CP consists of a number of work-stations and dedicated mini-computers. These devices are linked together through an ETHERNET local network. Each CP is connected to the others through existing networks (see section 2.4 below). Three types of functions are provided by the system: database management, operational situation management (including man/machine interface) and data/message communications. Accordingly, three types of hardware components are used:

(1) database management computers; BULL DPX family computers were implemented; it was decided to implement the system under UNIX with a view to using commercially available software; the DPX family was selected because it provides a wide range of machine performances with a reasonably good software portability;

(2) user interface devices; each of them is composed of a work-station and output devices such as printers and plotters; the system uses 68020-based UNIGRAPH graphic workstations;

(3) communication processors; they are composed of a UNIGRAPH work-station and modems including automatic dialing features when appropriate.

This architecture proved to be very convenient to provide the required system performance at each command level and to fulfill specific military requirements such as dynamic reconfiguration and gracefully degradable features.

2.2 Army and Corps Level

The architecture of an Army or Corps level CP is given in figure 2.1. BULL DPX/5000 RISC mini-computers are used instead of micro-computers as defined during the modeling phase due to the fact that the modeling was based on the assumption that a real time operating system would be used; given that UNIX task management system requires much more disk accesses than a real time operating system, it was necessary to use mini-computers to provide disk access facilities faster than those usually implemented in commercially available micro-computers.

Any hardware device used in the system is duplicated to allow possible reconfiguration in case of breakdown or destruction. For instance, there are two database management mini-computers and two communication processors. For the latter, they are used in a master/slave configuration. The master communication processor is managing all the network access points. It runs high level layer protocols for any data communication link

master or the slave communication processor depending on the physical link used. This allows for higher performance and more secure operation.

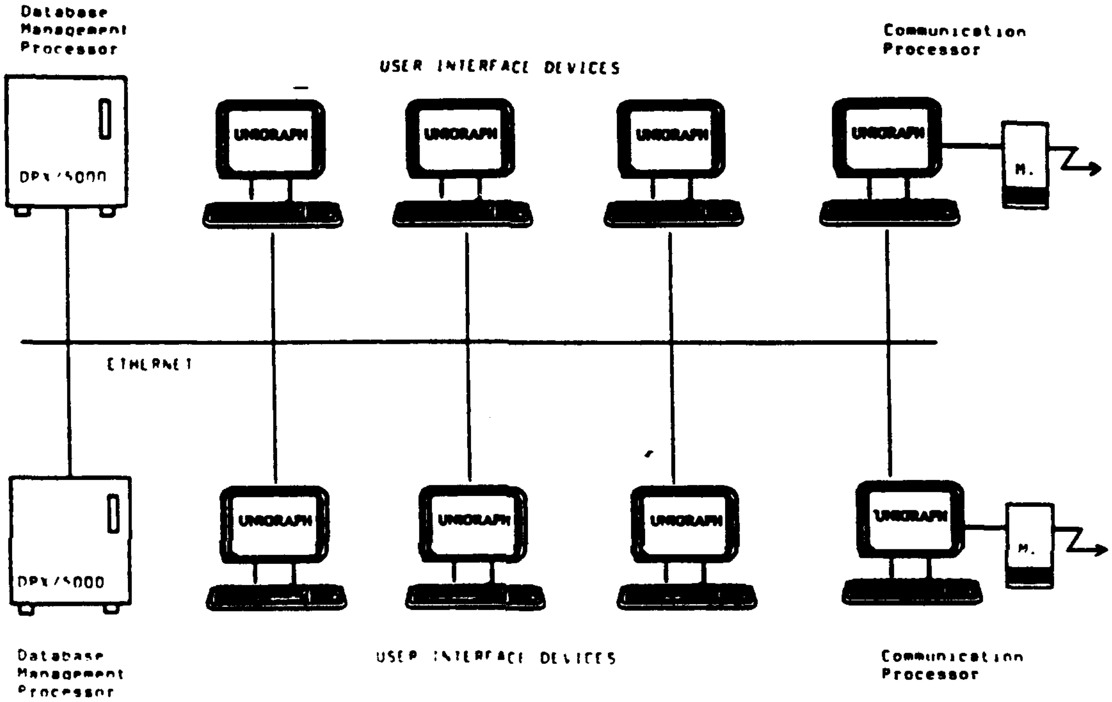


Figure 2.1

Corps Command Post Hardware Architecture

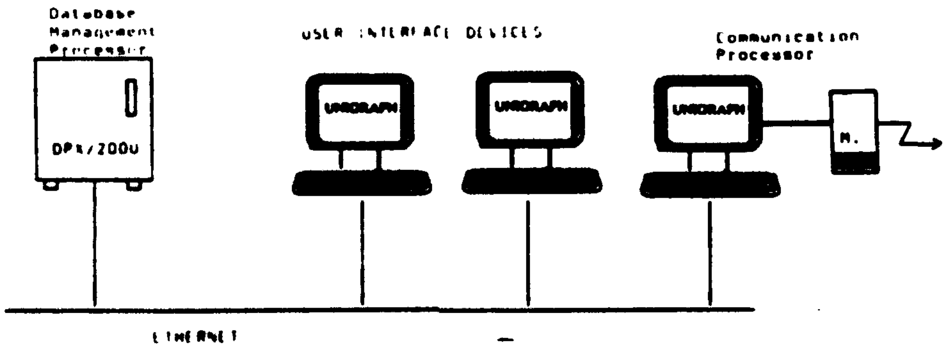


Figure 2.2

2.3 Division Level

The architecture of a Division level CP is similar to the architecture of a Corps level CP (see figure 2.2). However, attention had to be paid to the constraints imposed to the Division CP, namely the fact that a division headquarter may be deployed either in buildings or in shelters carried by trucks. For that reason, BULL DPX/2000 68020-based micro-computers are used as database management processors. Furthermore, it was decided not to duplicate the hardware components at the division level but to implement specific recovery procedures involving the alternate CP and a recovery database element (see section 4 below).

2.4 Communications and Interoperability

The RITA network is used as the primary communication channels between CP's (i.e. for external communications). The system can interface voice radio and cable access points (1200-2400 bps) as well as high speed data communication channels (48 kbps). Interfaces to the RITA message switching system (CAREME) are also provided. Specific modems were developed including automatic dialing functions when appropriate. These modems exist either as militarized or as commercial hardware.

Public military or commercial networks can also be used in different phases of the manoeuvre (e.g. military infrastructure network as RITTER in France or civilian data network such as TRANSPAC). In this case, for security reasons, the information is encrypted/decrypted using appropriate off-line encryption devices before/after their transmission through the non-secure part of the communication system (i.e. at the communication processor level).

Currently, the SICF is a central element of the C2 organization in the French Army. This means that the system will interface C3 systems above the Army level (national and NATO C3 systems), at the Corps level (ally C3 systems) and below the Division level (national C3 systems such as ATLAS for artillery and the Regiment Information System (SIR) for the other services).

A quadrilateral effort in co-operation with Germany, the UK and the US led to an interoperability demonstration this month (May 1990). It is intended to use the designed interface (based on X-400 message handling protocols) also for national purposes.

2.5 Military Issues

Given that the system is deployed in a rather adverse environment, it was decided to study and implement some hardware protection. Specific containers were designed to protect the hardware components when moving. Wheh the CP is deployed the containers are unloaded and front panels are removed so that the man/machine interface is accessible. Reinforced connectors were

is not clear what level of protection was actually required and further study is probably required to elucidate the actual environmental resistance of commercial hardware.

3. SOFTWARE GENERAL ARCHITECTURE

3.1 Application Host Software Package Concept

One of the main objectives was to use an evolutionary approach for software development. The rationale was twofold. First, budget constraints imposed to delay the specification and development of a number of user functions. Furthermore, by nature, military requirements evolve: new conditions (e.g. introduction of new systems such as the C2 system itself, or new weapons) imply changes to the C2 organization and methods which in turn imply evolutions of the C2 system.

3.1.1 Information Description Language

The basic design concept is that most of the work performed through an information system consists of a sequence of data representation transformations which comply to input/output formatting rules. For instance, operational data is sent to a CP as a formatted message (using specific NATO formatting rules). The content of the message is stored in a database using the corresponding database access language. Then the data are displayed on a graphic display using the corresponding graphic display commands, etc (see figure 3.1).

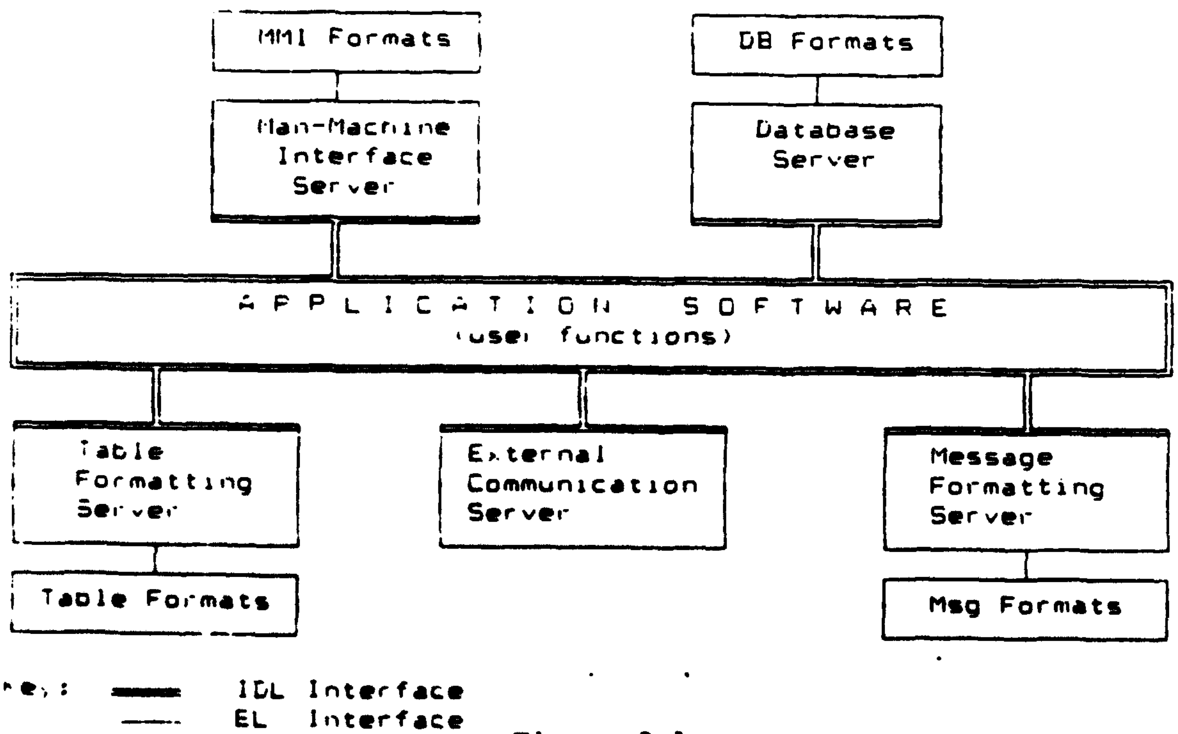


Figure 3.1

It is therefore possible to define a common information description language (IDL) which will be used:

(1) to convey information from one process to an other;

(2) to design a limited set of operations to manipulate information coded in IDL; these operations are for instance: read, write, initialize, etc;

(3) to design tools to generate automatically the codes used to transform the IDL into External Languages (EL), i.e. languages used for input/output (including the accesses to the database and the communication system); the generated codes are called servers.

3.1.2 Software organization

The IDL being defined, from a static point of view, the software consists mainly in a number of pieces of code or tasks, each of them transforming IDL to IDL or IDL to/from EL. These tasks must be executed in a given sequence to perform the required user functions. The application software implements this set of user functions.

It consists of a set of transactions. A transaction is a coherent set of processes called Transaction Processes (TP). A transaction may run several TPs in parallel but each TP is a sequential process (from a UNIX standpoint, a TP is an executable program).

The behaviour of each upper level entity (namely application and transactions) is described as a sequence of guarded commands ("if condition then action"). Conditions are lower level entity state variables. Actions are the exchange of commands and/or state variables with the upper level and lower level entities. An entity receives commands from the upper level entity and send state variables back; it sends commands to the lower level entities and receive state variables back. Executable programs are automatically created by a Transaction Process Engine (TPE) generator. Therefore, no software development is required at the application or transaction levels.

The basic IDL manipulation functions, the tools used to generate servers and TPEs and the servers themselves are included in the "Application Host Software Package" (AHSP). The TPs are called the "Application Software".

3.2 Application Host Software Package Design

The AHSP was developed using C language under UNIX System V. This choice was made at the very beginning of the design. The rationale to use UNIX was the large number of available software packages and the multi-user capability of this operating system. Accordingly, C language was used because the UNIX/C interface is straightforward which was not the case for any other languages such as PASCAL, ADA or LTR3 (French Military Standard Real Time Programming Language).

Three layers have been defined (see figure 3.2):

(1) at the bottom level are the standard packages, i.e. UNIX System V, GKS, X-WINDOW, TCP/IP, CLIO database management system (SYSECA), etc;

(2) the second level is designed to make the upper software layers configuration independent (i.e. to hide considerations of the actual hardware architecture of the CP from upper layer software); this level contains basic functions to manipulate the IDL and to manage application contexts, recovery points, inter-process communications and system supervision;

(3) the third level contains IDL/EL transformation codes; these codes may be associated to a physical device through the appropriate standard package (Man-Machine Interface Server through GKS (alphanumeric and graphic), Database Management System Server through CLIO, External Communications Server through TCP/IP and a X-400 Message Handling System) or they may be standalone (Message Formatting Service, Table Formatting Service).

For development and testing purposes, the AHSP also contains all the tools needed to generate and test the system, including dialog simulation tools to help specify the man-machine interface and on-line specific debugging tools. Note that the AHSP represents the "fixed" part of the software, i.e. the part of software which does not need to be modified or is generated automatically through development tools when new user functions are implemented.

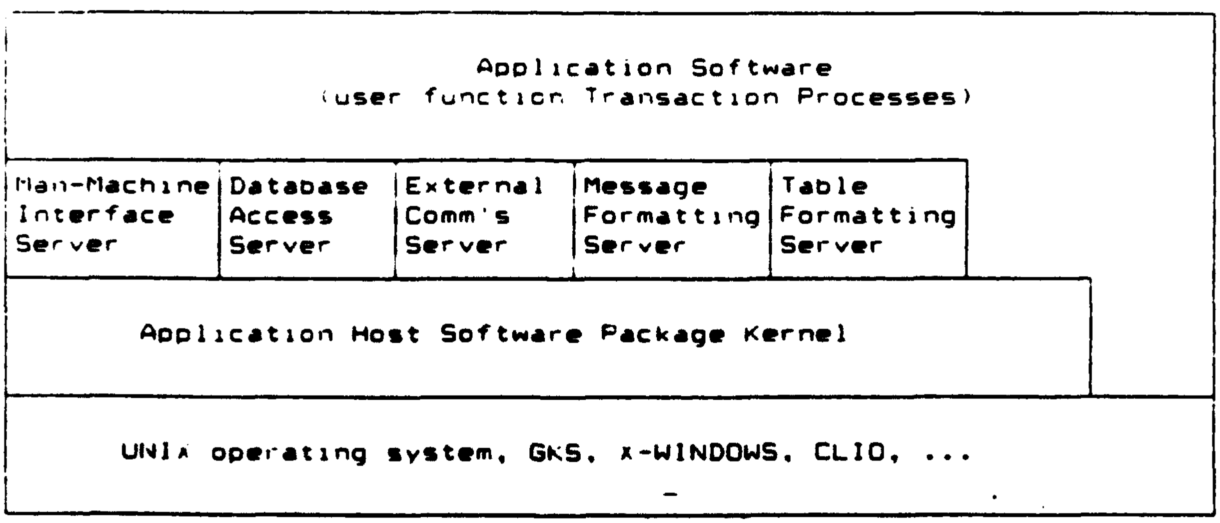


figure 3.2

3.4 Application Software Architecture

The application software consists of user function specific codes which are executed in conjunction with the AHSP code. To develop a new user function it is only necessary:

(1) to define extensions to the IDL (if required) and to run the AHSP tools to generate the associated codes (actually these tools generate the IDL manipulation functions, the IDL/EL transformer codes (servers) and a number of C "include" files which contain C data structure definitions to be used by the programmer when developing new TPs);

(2) to develop those TPs which are specific to the new function; these TPs will primarily consist of C functions which translate an IDL input file into an IDL output file; they contain the algorithmic part of the new functions (the actual data processing); various decision aids, including limited expert systems in the near future, may be implemented at this level;

(3) to create the tables used by the TPE generator to generate the corresponding transactions.

From a methodology standpoint, it is not possible to adopt the classic water-fall software life cycle model when this new software design approach is adopted. In fact, one cannot find a simple linear relationship between a user function requirement and its implementation in the system. Instead, this relationship is better represented by a matrix which relates user functions to TPs, given that one TP may be used in various user functions. Therefore, new software management procedures (and notably new software test and validation procedures) are to be defined in this context.

4. SPECIFIC FEATURES

4.1 Survivability Requirements and their Refinements

Survivability is a criterium of paramount importance for C3 systems. It is essential that information remain available even in case of breakdown and/or destruction. To achieve this goal, an appropriate level of hardware redundancy together with suitable data duplication procedures should be implemented. However, a fully duplicated distributed system is not technically feasible nor affordable. Trade-offs can be considered taking into account the specifics of military requirements and the military operating procedures:

(1) Usually, CPs are at least duplicated; however, given that these CPs are mobile and that they can work only when they are stationary, it is necessary to have three sites available to maintain duplication when one CP is moving (one moving, two stationary). This is the case at the Corps level but not at the Division level. When three sites are not available, it is possible to dupli-

to the same unit) using the RITA communication capabilities or to use the internal redundancy of a Corps CP to move that CP in two steps (one half moving, one half stationary).

(2) A CP is organized in cells (for instance: manoeuver, intelligence, etc..). Each cell is in charge of maintaining part of the information available at the CP: the enemy situation is maintained by the intelligence cell, the friendly forces status is maintained by the manoeuver cell, etc. Therefore, it is possible to define sub-sets of data each of which "belongs" to one cell: only a given cell (i.e. a known set of work-stations) can modify the data in a given sub-set. These sub-sets can be used as granules of database distribution in order to simplify access control.

(3) The lifetime of most of the information maintained at a Corps or Division CP is long compared to the C3 system response times. This means that it is admissible to "loose" processed data provided that the source data remain available. This means in turn that some delay in the duplication of processed data is possible but that the message handling system should be very robust.

(4) It is possible to define priority user functions based on the tactical situation. These functions should survive a catastrophic breakdown or even destruction. However, priorities change. Thus, totally automatic reconfiguration is not adequate but system administration functions should be made available to manage the system. A specific work-station is given this function.

4.2 Duplicated/Distributed Database Management

Database duplication in a CP is achieved by mirror database-like mechanisms. But, instead of using two disk drives connected to the same CPU, the facility is implemented using two separate CPUs and disk drives. Data exchanges are supported by the CP's ETHERNET local network.

Database updates are executed first on the source database server. If the updates are executed properly, they are batched and will be sent to the image database server only when a recovery point is reached by the client transaction. This guarantees data coherence and simplifies error recovery. When an error is detected, an error report is sent to the System Administration Workstation (SAW) where system management functions are available to restore a correct state (see Section 4.4).

Given that the source server function consumes more computing power, the source/image server functions are distributed among the two available database management processors. One processor will be a source server for one data sub-set and image server for another. This can be dynamically changed by the SAW without system interrupt (in fact, database accesses are suspended during a short period of time which is transparent for the end

4.3 Command Post "Step-Up"

The Command Post "step-up" consists in deactivating the "active" (main) CP and activating at the same time the Alternate CP. It is a complex process due to data coherence and time constraints. Attention should be paid to the fact that when the alternate CP is moving, database updates will not be available (nor processed) at the alternate CP. Hence, it is required first to update the alternate CP database and then to maintain information at both CPs. All data transfers use the RITA high speed data links (48 kbps) to reduce the transmission delay. However, it was shown during the system modeling phase (see section 1.2) that the limiting factor was primarily the disk access load at the alternate CP. It is assumed that the databases at both CP are identical at the beginning of the exercise (or war). Thus, the step-up process requires three steps:

(1) The source database of the alternate CP is updated using the image database of the active CP. Updates of the image database are suspended and stored in a separate file (journal). Messages are received and processed at the active CP. Even though the alternate CP is operating, database user updates at this CP are not allowed;

(2) the image database of the active CP is re-activated. The update journal is periodically sent to the alternate CP. Messages received at the active CP are automatically relayed to the alternate CP (source information is therefore available immediately, and processed information with some delay); database user updates are not allowed at the alternate CP;

(3) The message send/receive function is suspended at the active CP. The last update journal is sent to the alternate CP and the active CP is disconnected from RITA. Upon receipt of the update journal, the alternate CP is disconnected from the RITA network and reconnected as the new active CP. From this point on, it will receive all messages sent to the active CP. This is totally transparent to its addressees. The old active CP can be reconnected to RITA as alternate CP or be dismounted and move.

Phase 2 of this process is performed continuously with a third element (for instance the rear Corps CP) in order to provide instantaneous step-up capabilities if the main CP is suddenly unavailable. In this case, the step-up process can be continued or re-initiated from this third element.

4.4 On-line Configuration/Reconfiguration Management

Specific features are implemented in the AHSP to support on-line system configuration management:

(1) One key element in configuration management is the gathering of error messages to help evaluate the system

damages and select appropriate recovery actions. All error messages are routed to the SAW. This function can be allocated to any work-station in the CP.

(2) System management functions are based on the concept of Software Logical Sets (SLS). Software logical sets contain a number of transactions. Each of these sets corresponds to those user functions which are needed by a specific operational user (such as intelligence cell transactions). A file back-up device is linked to each SLS. Recovery point information will be stored by the transactions or TPs on that device. SLS are installed as a whole by the configuration management functions so that all the transactions belonging to the SLS are executed on the same physical machine.

(3) the AHSP provides automatic routing of inter-process communication messages. To do so, only logical addresses are defined at compile time; physical/logical device correspondence tables are maintained through the SAW at run time and are used by the AHSP for message routing.

The following user functions are provided at the SAW:

(1) Error messages display. A catalogue of possible recovery actions is provided to help the user find the solution corresponding to the detected error;

(2) Mapping SLS to physical machines. The mapping can be changed dynamically allowing any kind of system configuration/reconfiguration to be performed with little system technical knowledge.

(3) CP "step-up" management functions. Given the complexity of the process, aggregate functions are provided to perform a CP "step-up".

5. CONCLUSIONS AND RESEARCH RECOMMENDATIONS

The system was fielded beginning in 1988. After a maturation period, it is now considered as very helpful by the users and is used routinely during military exercises. Further software versions are fielded regularly to provide new user functions.

From the development standpoint, the initial modeling phase together with the experience of the previous system test-beds proved to be very useful to help identify possible limitations and determine the system architecture concepts. The main conclusions drawn from the modeling phase appeared to be valid even though the actual system design was quite different (due to the fact that commercially available hardware and software were used). Such a modeling effort is therefore strongly recommended for any new development. However, further research is required to improve overall distributed system performance evaluation model-

ing, with a view to guiding the design process, i.e. to answering the question "What is the most cost effective architecture to provide a given level of performance?"

One major issue has been the design of the AHSP. Although the basic principles (IDL, servers, etc) were stated at the very beginning of the system design process, the lack of design and development tools made the development process difficult. A lot of effort was unduly spent in designing and developing the software. However, it is argued that this type of "parameterized" software is the most appropriate state-of-the-art technology to fulfill evolutionary system development needs. Furthermore, even though most C2 systems are nationally specific, opportunities for co-operative C2 system development can be found for the AHSP.

Hence, research work is required in the area of development methods for this kind of system. Research should also be directed toward the study of the ways new technology (either hardware or software) can be incorporated in an existing system. This would be extremely helpful in the context of evolutionary system design.

Validation tools are also required to ensure that survivability functions are correctly designed and implemented. Even though there is a strong presumption that the implemented reconfiguration functions fulfill the survivability requirements, there were no methods to evaluate and to validate this aspect of the system.

REPORT DOCUMENTATION PAGE

1. Recipient's Reference:	2. Further Reference:
3. Originator's Reference: AC/243(Panel 11)TP/1	4. Security Classification: UNCLASSIFIED/UNLIMITED 5. Date: 15.04.91 — 6. Total Pages: 13

7. Title (NU):

C3 for the 90s - New Ideas in Survivability

8. Presented at:

AC/243(Panel 11) Symposium on Military Information Systems Engineering - RSRE, Malvern, UK - 8-10 May 1990

9. Author's/Editor's:
Mr. Richard A. Metzger, Mr. Carl A. DeFranco Jr.

10. Author(s)/Editor(s) Address:
RADC/COTD
Griffiss AFB
NY 13441-5700
United States

11. NATO Staff Point of Contact:
Defence Research Section
NATO Headquarters
B-1110 Brussels
Belgium
(Not a Distribution Centre)

12. Distribution Statement:

Approved for public release. Distribution of this document is unlimited, and is not controlled by NATO policies or security regulations.

13. Keywords/Descriptors:

COMMAND AND CONTROL, DISTRIBUTED SYSTEMS, INTEGRATION, REPLICATION, REDUNDANCY, SURVIVABILITY

14. Abstract:

This paper uses a new approach to achieving survivability for Command and Control systems by application of new technologies in distributed computing and adaptive communications. By employing redundancy through replication and distribution, and using advances in Distributed Information Systems, enhanced probability of functional survival is achieved. The cost of replication is balanced through reduced manpower. Recent and on-going research and development efforts are discussed to illustrate the advances available.

C3 FOR THE 90's - NEW IDEAS IN SURVIVABILITY

Richard A. Metzger *
Carl A. DeFranco Jr. **

Table of Contents

* Mr. Metzger is a Supervisory Computer Scientist and
Chief, Distributed Systems Branch
Command and Control Directorate
Rome Air Development Center
Griffiss Air Force Base, NY, USA

** Mr. DeFranco is an Electronic Engineer and
Center Program Manager
Distributed Systems Branch
Command and Control Directorate
Rome Air Development Center
Griffiss Air Force Base, NY, USA

1. INTRODUCTION

1.1 In a hostile environment, a key capability of any military system is survivability, which has been both a requirement and an elusive goal for Command, Control, and Communications (C3) systems for several decades. With military force mobility on the increase, C3 system survivability in the 1990's will take on new urgency. The driving functional requirements will be for dynamic response to a rapidly changing threat scenario. Major elements in providing this capability are rapid access to updated intelligence, adaptive target selection and mission planning, and a flexible C2 system to provide responsive force management. Critically important will be the need to sustain this capability through a hostile encounter. Traditional methods of providing survivability for C3 systems have been to harden and shield the facilities housing them. During the 90's, replication and dispersion will replace hardening as the primary method of achieving survivability. Systems that can dynamically recover and restore functionality among surviving nodes of geographically dispersed clusters will be the architectures of choice.

1.2 A series of research efforts are underway to bring the above capabilities to reality, addressing a number of major issues in the areas of distributed information processing systems and highly survivable communications systems. Intelligent resource management through distributed operating systems provides key capabilities in fault tolerance and reconfigurability. Uniform access across multiple computers provides the foundation for distributed database management which, in turn, provides concurrency and integrity for multiple users across multiple databases. Reliably linking command center assets in a dynamically changing topology with sporadic connectivity will require major advances in adaptive communications. Opportunities abound for application of expert systems and knowledge bases to speed the decision process. All of these areas are being agressively pursued and promising results are emerging.

2. BACKGROUND.

Although the principles that apply to survivability must be considered for both strategic and tactical warfighting capabilities, it is clear that application of these principles differs for the actual arena. The geographic domain under consideration differs significantly, the extent of the goals are not equivalent, and the time response may cause major differances. However, the classic methods of making a system survivable have been centered around making the elements physically stronger to withstand damage and if that is insufficient, to provide redundant copies to replace the lost elements.

2.1 GENERAL REQUIREMENTS FOR SURVIVABILITY

2.1.1 Physical survival involves preserving the existence of an actual physical resource, e.g. facilities and people. To do so generally requires physical action such as removal of the asset from the destructive environment or hardening of the resource against destruction through physical protection methods, or both.

2.1.2 Functional survival is concerned with preserving a capability to perform rather than protecting a specific resource. In other words, if we wish to insure our capability to do tactical planning, we ensure that the planning function survives, rather than a planning center. This would normally involve replication and dispersion of assets to provide survival by numbers rather than hardness.

2.1.3 Modes of degredation will also affect system survivability, depending upon the system's fault tolerance or sensitivity to failures within the system. Although the distinction may not be black and white, we will focus on two principal modes here:

1) Catastrophic - the failure mode that exhibits a step transition from fully operational to totally non-functional. Obviously, a system subject to functional failure for minor subsystem problems will not provide acceptable probability of survival.

2) Graceful - the failure mode in which the effects of subsystem failures on overall functionality is minimized until some lower bound of capability is reached. Careful system design and redundancy can greatly improve resistance to faults and failures.

2.1.4 Connectivity is a general measure of the ability of system elements to pass or exchange information. A number of factors and parameters are used to describe and measure connectivity within a system.

1) Topology- which defines routes of connectivity between set of nodes forming the system

2) Links- physical channels between nodes of the topology which provide the transmission media and are characterized by:

a) Bandwidth- data per unit of time transmitted over a link

b) Delay- time between sending and receipt of data on a link

3) Protocols- which control and specify the physical and logical flow of data across the links of the system

4) Throughput- the aggregate measure of data transiting the system per unit time based upon all of the above factors as well as the sending and receiving processes

5) Control and Management - the tools by which all of the above elements are observed, measured, and controlled to achieve optimum performance of the system, or achieve specific objectives.

3. SPECIFIC APPROACHES TO SURVIVABILITY

3.1 Hardening. Conventional protection methods have generally depended upon two basic techniques: hardening and distance. The distance method involves placing valuable assets far to the rear of battle, reducing probability of destruction by weaponry. With modern missile technology, this method is far less effective than in the past. The hardening method relies on current technologies and improvements in materials science to provide necessary structural stability against detonating weapons and the corresponding impulse loads on buildings. Substantial structural reinforcement, underground construction, and blast deflection techniques are common. In addition, isolation of internal structures from the external building by large springs provide additional protection against shocks. To To protect against nuclear, chemical and biological hazards carried from the external environment, decontamination facilities are provided to remove contaminants from personnel entering the facility, and to purify the air and water used internally. While these methods provide reasonably effective protection against conventional warfare techniques, they are becoming extremely expensive and may not provide effective resistance to nuclear effects.

3.2 Redundancy. The concept of alternate command centers is very old, but the approach has been used mainly to replace a destroyed primary facility. Duplication of facilities attempts to provide functional survival by creating and maintaining sufficient copies of a C3I system or subsystem to insure that total destruction is improbable. Often, the alternate facility is often only minimally capable compared with the primary. Even when automated information systems are included in command centers and facilities, the backup capability of alternate centers remains limited. Given the rapid advances in the speed and power of modern computers, coupled with dramatic reductions in cost, size and energy input, there is no reason for alternate facilities to be any less functional than primary ones.

3.3 Employing Redundancy. Efficient use of redundancy as a technique involves integrating two separate concepts: physical replication and dispersal, and logical unification.

3.3.1 For protection, physical dispersal of replicated copies is essential, since the two main objectives are to insure survival of the functions contained within the asset and to decrease the probability of destruction by enemy action to a very low value. For example, a determined foe might develop a capability to destroy a certain command facility with a probability of 0.80. If we replicate the facility into four copies, the probability of all four being destroyed, i.e. loss of that entire function, is only 0.41; our foe would require a fourfold increase in weaponry, and his probability of success is cut in half.

3.3.2 Logical unification may be achieved in two major forms, networked systems and distributed systems. Networking is a reasonably mature technology that provides mechanisms for exchanging data across some form of multiple user connectivity. Users generally require some knowledge of other entities on the network in order to establish the data interchange. Distribution builds upon networks, and extends the concept by managing the global (to the network) tasks of addressing and resource allocation across the network. To the user, the network of resources becomes one large virtual computing system, and the concept of geographic location vanishes.

3.3.3 Virtually every Command and Control system of the future will be critically dependent upon high performance computing systems to provide commanders with immediate access to vast quantities of data. Reliance on these systems will extend from remote sensors acquiring and fusing data under computer control, to planners using sophisticate tools to

generate options and plans, to the execution and monitoring of the mission itself. In addition to these operational functions, the information handling system must exhibit the same level of survivability as other elements of the system. Distributed Information Systems (DIS) offer the potential for major increases in survivability through dynamic reconfiguration of the allocation of processing resources around lost elements.

3.3.4 This increased survival does not come without cost. In our example above, we must create four complete copies of the resource to be preserved. In a manpower intensive system, this may involve considerable expense and require personnel not always available. In this realm, reduction of manpower through automated information systems provides the fulcrum on which to balance manpower and equipment costs. The agressive use of computer automation coupled with the rapdily advancing technology in artificial intelligence and knowlege engineering promise to provide the reduction of critical manpower.

3.3.4.1 Communication is required to support distributed systems, both for information passing and for control and management. Such connectivity is required to provide a logical path for information between any two elements that are or could be integrated into a functional system or subsystem. Given the added rquirement for system survivability, we must provide support for operation under less than ideal conditions.

3.3.4.1.1 Reconstitution/reconfiguration are required to heal damage to the connectivity or to add additional elements to the existing system. Such capability must include some intelligent decision-making process that directs the reconfiguration into the optimum connectivity for the system. The term often used to describe this function is the Intelligent Communications Controller. Current efforts include research into methods that couple the computing resource allocation function with the communication resource allocations. This coupling provides the system management function with insight into process communication requirements that can be used to crate an optimized configuration for current operational demands.

4. CURRENT PROGRAMS

4.1 Multi-Media Communications

4.1.1 Several programs are underway to develop a robust connectivity by combining transmission media into a unified system. One current program at RADC is known as Multi-media Communications Capability or M2C2. The long-term objective is to combine diverse physical transmission media such as multi-band RF, fiber optics, wirelines, and satellites to produce a transmission system with enough physical endurance to preclude a total loss of connectivity. Management and control capability will be integrated into the system. A key physical characteristic is small size and high mobility for tactical situations.

4.1.2 The present work has focused on creating a multiple radio assembly covering the RF spectrum from HF to UHF, a controlling computer system, and the software required to monitor transmission requirements and extend the channels over which information can flow by creating alternate paths from available links, or by combining them into parallel channels for higher throughput. Such a system provides link survivability in a synergistic fashion by combining systems that display differing sensitivities to interfering energies. An example is the time-varying effects of nuclear atmospheric effects which disrupt VHF/UHF bands quickly but over relatively shorter periods than HF bands, which degrade more slowly and for longer times.

4.1.3.1 The current system accepts data for transmission, and adaptively creates packets for transmission, packet size being dependent upon link quality. Link quality is judged by the number of retransmissions required for success, and packets are dynamically adjusted during transmission. The RF channel selection begins with the highest speed channel available, and shifts downward in capacity as the RF media degrades. Link transmission speeds range from 16 kilobits per second over UHF and VHF channels down to 2400 bit per second over HF. Because transmission is packetized, channel failure affects only the packet in progress. After reestablishing the link on another radio, data tranfer continues from the last successful packet. A series of experiments indicates that information transfer can be considered error free. Thus, for the individual link, the system provides graceful degredation, taking advantage of other channels (and eventually other transmission media) that display differing effects from interference.

4.1.3.2 The original prototype system was built from available radio assets, and used the Zenith 248 computer, and IBM PC-AT compatible, as the system controller. Current versions use U.S. military radios and VME-bus 68030 microprocessors running Unix as a hardware base. This new configuration allows simultaneous transmission on a point-to-point link, and provides a store-and-forward capability as well as higher processing speeds. Data files ranging from simple text files to digitized video have been passsed across the system. In addition to the laboratory facility, two additional assemblies have been installed within vehicle mounted shelters. The initial prototypes have been so successful that requests for support from other programs is causing diffculty in scheduling system availability.

4.2 DISTRIBUTED INFORMATION SYSTEMS

4.2.1 The key to providing Distributed Information System (DIS) functionality is the resource management capability provided by the system level software called the distributed operating system (DOS). The DOS provides the global integration environment which makes the collection of computers in the DIS appear as a single cohesive computing environment. Within this DOS environment all entities (directories, files, processes, etc.) have a unique identity assigned by the DOS. This global name space spans all of the machines and allows the DOS to provide the user with location transparent access to any of the objects, without regard to which machine the request originates from, or on which machine the data may be located. Similarly, the system can readily move files or processes among the machines since a uniform environment exists across all of the elements. If an application progrm has need of a particular file it can provide the identifier for the file to the system, and the system will know from the identifier which computer in the configuration owns the file, and a call for transport or copy of the file can be made.

4.2.2 This ability to automatically move files and processes among the nodes of a DIS has significant potential for enhancing survivability. Critical data files or application modules can be automatically replicated and maintained current at a remote host. In the event of the loss of a primary copy the remote copy can be activated as primary, and with a "rolling" replication scheme a replicated copy will always be available.

4.2.3 There are numerous trade-off's that must be considered in designing any automatic replication scheme. Maintaining consistency in a replicated database has been the subject of extensive research over the last fifteen years. While a solid theory exists, the implementations are plagued by performance degradation resulting from the synchronization requirements. Newer insights based upon the ideas of "weak consistency" show promise for reducing the overhead by relaxing the requirements of strict consistency based upon serializability by taking advantage of application specific characteristics of the data.

4.2.4 To fully realize the benefits of distributed systems for survivability, dynamic process replication must also be developed which does not rely strictly upon checkpointing. A partially completed process that is terminated must be restarted from a stable state usually represented by a checkpoint. If the time between checkpoints is long, then time to recover to the terminated point is large. If finer grained checkpointing is used then the overhead it introduces can significnatly decrease performance. Mechanisms are being developed which can replicate processes togehter with their in-process state information which will provide much greater flexibility in replication and point the way toward dynamic process migration.

4.3 Planning in a Distributed Computing Environment.

4.3.1 Known as PDCE, this program focused on investigation of distributed computing architectures containing planning systems, and the system parameters affected by the system design. Under contract with the Rome Air Development Center, Advanced Decision Systems of Mountain View, California, USA, focused on the primary goal of survival of function as discussed above, i.e. survival of the capability to create a plan, rather than the survival of a planning cell as the important result. In addition, investigation of measures of performance were made for systems utilizing checkpoint-restart as a survivability technique, as well as negotiation among computing entities for task execution.

4.3.2 Three architectures were proposed as providing some form of functional survival:

1) A small number, less than 5, of highly capable replicated planning nodes, each operating autonomously, but with knowledge of the others. A primary node is selected to perform all required planning tasks. Others provide standby capability.

2) A moderate number, up to 50, of fully capable replicated planning nodes operating in a cooperative fashion, sharing various tasks of planning. Each node has sufficient power to do all required computing, but may tasks are shared.

3) A larger number, over 50, of partially capable nodes, with limited and unequal computing power. The system creates a single virtual planner by efficiently utilizing the capabilities of the nodes to complete a plan.

4.3.3 Given that other programs were focusing on the first architecture, the focus fell to architecture two as within reasonable technical and cost bounds for accomplishment. Based on a functional model of a US strategic system, the planning system was broken into operations and required data files. The design makes to following assumptions:

1) Each node has enough computing power to complete a task.

2) Each node has the current copies of all data required.

3) The planning can be abstracted into nine basic "roles," defined by the database categories involved, e.g. targeting, assignment of weaponry, etc.

4.3.4 The model, known as the Network Reconsititution Simulator (NRS), simulates a geographically wide distribution of nodes across the continental United States, and presumes that a surprise attack will randomly destroy some fraction of the nodes. In order to reestablish planning functionality, a reconstitution algorithm is executed in two stages. Prior to disruption, the system computes the connectivity from each node to its neighbors, and heuristically determine the best match of function to node, based upon available throughput and interprocess communications requirements. After disruption, which includes random loss of nodes, surviving nodes execute a broadcast mechanism and build tables of connectivity, each then selecting the most appropriate role or function based upon its connectivity. An optional system function tested added the ability of nodes to negotiate their roles with other nodes. This extension of reconfiguration, based upon initial tests, can improve planning performance as much as 75 percent, and provides a much more graceful mode of degredation.

4.3.5 Coupled with the NRS tests, measurements were made to determine the effect of various scheduling algorithms used in operating systems to evaluate their efficiency. Results indicate that the scheduling algorithm selected is dependent upon the types and sizes of the processes to be run, the existence of special purpose processors, and the homogeneity of the majority of processors. Analysis was also done on the costs of incorporating a checkpoint-restart capability into the operating system that allows migration of processes between hosts within a system. Such a capability provides enhanced reliability by improving functional survival, but at some cost in time. The studies included examination of the checkpointing frequency using a binary tree and a vector computation. No specific recommendation was developed, but rather a set of data that can be used in evaluating future work.

4.4 The Survivable Adaptive Planning Experiment is a program aimed at demonstrating the applicability of advanced computing and communication technologies toward creating a deployable, survivable strategic planning system that functions in a highly stressed environment. Currently in its Exploratory Development phase, the second of a four phased approach, the long term goal is to fully demonstrate the planning software in a multi-node configuration, with simulated environmental stresses. Whereas traditional strategic planning has been a highly centralized batch process, with control and parameters often embedded directly within executable code, the new system design draws upon object oriented design principles, separating the system into four identifiable areas:

1) Knowledge based functional control

2) Algorithmic processes

3) Data and database management

4) Inter and intra-node communications

4.4.1 Underlying the system design is the creation of a fully distributed computing capability, both within the basic node, and among nodes in an overall network. This distribution is being based upon the Cronus Distributed Operating System, developed over the past eight years by RADC in conjunction with BBN Laboratories in Cambridge, Massachussetts, USA. Cronus provides both the distributed and the object oriented environment for the software engineering.

5. CONCLUSIONS

5.1 New Choices. The state of the art in both communications and data processing have provided new options for achieving higher levels of survivability in future command and control systems. Hardening is no longer the only, or for that matter, the best way to achieve endurance. A new alternative is to replicate copies of the critical elements that can become active when the necessity arises. Recent advances have made this redundancy in distributed copies possible. While there is additional overhead associated with maintaining the consistency and concurrency of the replicated copies, the inclusion of the capability which provides that function improves overall system performance. It allows the system to reallocate resources for better performance and fault recovery. The ultimate benefit is the assurance that the system will not fail catastrophically and that as long as some minimum set of resources remains, critical mission functions will be executed.

5.1.1 The achievement of these goals requires the additional development of technologies that are now in the feasibility stage. To support the multi-cluster distributed information system that is needed, communications must be available both in hostile and non-hostile situations. This capability will be provided by the multi-media communications systems, which not only utilize multiple transmission media, but also can intelligently manage the data transmission among them. In a commensurate way, the computing elements that connect through the communications nodes to form the complete distributed information handling structure must be capable of adaptively allocating processes and data among the nodes. Basic feasibility of these concepts has been demonstrated and it is reasonable to expect their availability for operational usage within the next several years.

5.2 Integration. The key to the successful employment of all these advanced technologies is the skillful integration of these systems. A major aspect to speeding the development and transition of the technology is the use of rapid prototyping. As the technologies are demonstrated as feasible, the integration of them into testbeds that are accessable to both the developer and the user is an excellent vehicle to obtain early feedback into the R&D community. While there will be limitations in both functionality systems point of view, and interactions with other elements of the system can be identified. The use of this "build a little, test a little" strategy can provide a beneficial evolutionary path for maturing of the technology.

Finally, as the technology approaches full scale development, the application of system engineering principles in both hardware and software specification, engineering, and design is essential in quickly bringing operational capability into the real world realizing the full potential of modern information processing technology.

References

Berets, J.; Sands, R.; "Introduction to Cronus", Technical Report RADC-TR-89- 151, Vol IV, Rome Air Development Center, Griffiss AFB NY, 13441-5700

Cromarty, A. et.al., "Reconstitution, Reconfiguration, and Knowledge-based Routing in a Heterogeneous Distributed Computing Environment," Technical Report ADS-TR-3112-02, Advanced Decision Systems, Mountain View, CA

Cromarty, A. et.al., "Dynamic Adaptive Resource Management for Real-Time Distributed Planning,", Technical Report ADS-TR-3191-1 (Not Released for Publication at this time), Advanced Decision Systems, Mountain View ,CA

DRG DOCUMENT CENTRES

NATO does not hold stocks of DRG publications for general distribution. NATO initiates distribution of all DRG documents from its Central Registry. Nations then send the documents through their national NATO registries, sub-registries, and control points. One may sometimes obtain additional copies from these registries. The DRG Document Centres listed below can supply copies of previously issued technical DRG publications upon request.

BELGIUM
EGM-JSRL
Quartier Reine Elisabeth
Rue d'Evere, 1140 Bruxelles
Tel:(02)243 3163, Fax:(02)243 3655

CANADA
Directorate of Scientific Information Services
National Defence Headquarters
MGen. George R. Pearkes Building
Ottawa, Ontario, K1A OK2
Tel:(613)992-2263, Fax:(613)996-0392

DENMARK
Forsvarets Forskningstjeneste
Ved Idrætsparken 4
2100 København Ø
Tel:3927 8888 + 5660,
Fax:3543 1086

FRANCE
CEDOCAR
00460 Armées
Tel:(1)4552 4500, Fax:(1)4552 4574

GERMANY
DOKFIZBw
Friedrich-Ebert-Allee 34
5300 Bonn 1
Tel: (0228)233091, Fax:(0228)125357

GREECE
National Defence Headquarters
R+T Section (D3)
15561 Holargos, Athens
Tel: (01)64 29 008

ITALY
MOD Italy
SEGREDIFESA IV Reparto PF.RS
Via XX Settembre, 123/A
00100 Roma
Tel:(06)735 3339, Fax:(06)481 4264

THE NETHERLANDS
TDCK
P.O. Box 90701
2509 LS Den Haag
Tel:(070)3166394, Fax:(070)3166202

NORWAY
Norwegian Defence Research Establishment
Central Registry
P.O. Box 25
2007 Kjeller
Tel:(06)80 71 41 Fax:(06)80 71 15

PORTUGAL
Direcção-General de Armamento
Ministério da Defesa Nacional
Avenida da Ilha da Madeira
1499 Lisboa
Tel:(01)610001 ext.4425, Fax:(01)611970

SPAIN
Ministerio de Defensa,DGAM
SDG TECIN, C/ Arturo Soria 289
28033 Madrid
Tel:(91)3020640, Fax (91)3028047

TURKEY
Genelkurmay Genel Plan Prensipler
Savunma Arastirma Daire Baskanligi
Ankara
Tel:(4)1176100 ext.1396, Fax:(4)1250813

UNITED KINGDOM
DRIC.
Kentigern House, 65 Brown Street
Glasgow G2 8EX
Tel:(041)224 2435, Fax:(041)224 2145

UNITED STATES
DTIC
Cameron Station
Alexandria, VA 22304-6145
Tel:(202)274-7633, Fax:(202)274-5280

DEFENCE RESEARCH SECTION
NATO HEADQUARTERS
B 1110 BRUSSELS
BELGIUM
Telephone [32](2)728 4285 - Telefax [32](2)728 4103
(not a DRG Document Distribution Centre)